A hardware MP3 decoder with low precision floating point intermediate storage

Andreas Ehliar, Johan Eilert

LiTH-ISY-EX-3446-2003
Linköping 2003

A hardware MP3 decoder with low precision floating point intermediate storage

Master's Thesis

in **Computer Engineering**,
Dept. of Electrical Engineering
at **Linköpings universitet**

by **Andreas Ehliar, Johan Eilert**

Reg no: LiTH-ISY-EX-3446-2003

Supervisor: Mikael Olausson
Examiner: Dake Liu
Linköping, 2003

Abstract

The effects of using limited precision floating point for intermediate storage in an embedded MP3 decoder are investigated in this thesis. The advantages of using limited precision is that the values need shorter word lengths and thus a smaller memory for storage.

The official reference decoder was modified so that the effects of different word lengths and algorithms could be examined. Finally, a software and hardware prototype was implemented that uses 16-bit wide memory for intermediate storage. The prototype is classified as a limited accuracy MP3 decoder. Only layer III is supported. The decoder could easily be extended to a full precision MP3 decoder if a corresponding increase in memory usage was accepted.

Contents

1

Introduction

1.1 Purpose of this work

MPEG-1 layer III is well understood, both on desktop systems and in embedded systems. Embedded systems usually use fixed point arithmetics, whereas decoders for desktop systems can be implemented using either fixed point or IEEE floating point arithmetics.

The purpose of this thesis project has been to evaluate if it is practical, in terms of sound quality, to use a 16-bit wide data memory to store the intermediate sample values and other data during the decoding process. There are several reasons to minimize the word length of the data memory. By reducing the size of the memory and the size of the multiplier hardware, power consumption and chip area is reduced. A floating point format was used for achieving the required dynamic range with a modest word length.

1.2 Report outline

General background information about perceptual audio coding and the MP3 standard is given in chapter 2.

Chapter 3 contains results from the floating point precision prestudy.

The hardware architecture of the implemented MP3 decoder is described in chapter 4.

The development tools that were implemented to facilitate programming the hardware are described in chapter 5 and the decoder software is described in chapter 6.

Decoder benchmarks and profiling statistics are found in chapter 7.

Chapter 8 summarizes the FPGA prototype.

Chapter 9 contains a summary of the results and chapter 10 lists possible future improvements.

1.3 Acknowledgements

We would like to thank our examiner Dake Liu and our supervisor Mikael Olausson for the opportunity to work with this interesting and challenging project.

We would also like to thank our opponents Johan Borg and Gernot Ziegler for providing comments and feedback on our report and Maria Axelsson for additional comments.

2

Background

This chapter gives a brief introduction to the basics of perceptual audio coding, followed by an equally brief overview of the MP3 standard. A much more in-depth introduction to perceptual audio coding and various audio coding standards is given in Perceptual Coding of Digital Audio [1].

2.1 Perceptual audio coding

Perceptual audio coding is based on the fact that the human ear and auditory system extracts and uses much less information from the heard sound than is available. It is therefore often possible to remove or change some components in a sound without any *noticeable* difference from the original sound.

2.1.1 The masking effect

Research has resulted in *psychoacoustic models* that describe which parts of a sound that are actually heard (by humans) and which parts that are not discernible.

Apart from the well known fact that humans in general cannot hear frequencies above 20 kHz, there are other interesting properties of the ear known as simultaneous *frequency masking* and non-simultaneous *temporal masking*.

Masking means that one sound has become inaudible because of the presence of another sound (the masker). Frequency masking occurs when

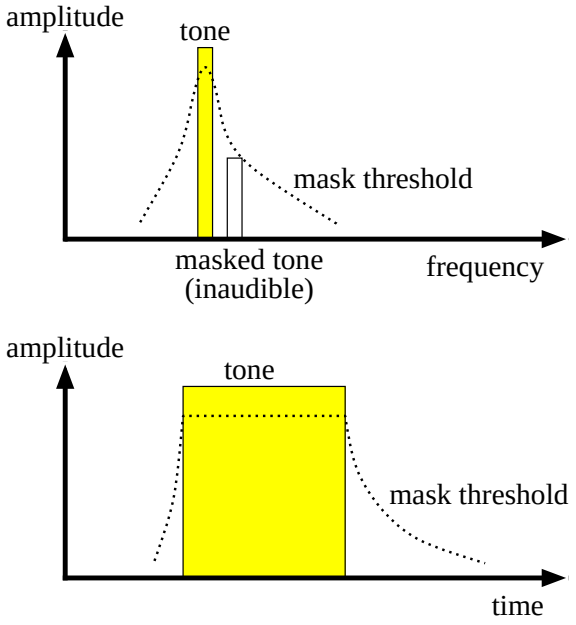


Figure 2.1: Frequency masking (top) and temporal masking (bottom).

a loud sound masks softer sounds that are close in frequency. Temporal masking occurs when a loud sound begins to mask a soft sound *before* the loud sound is heard. The temporal masking also continues a moment after the loud sound has disappeared. Figure 2.1 gives a graphical view of the masking.

2.1.2 Critical bandwidth

The nature of frequency masking makes it convenient to introduce the concept of *critical bandwidth*. The critical bandwidth increases nonlinearly with frequency and it determines the slope of the masking thresholds introduced by tones and noise. The *Bark* unit corresponds to the distance of one critical band.

2.1.3 Quality measurements

With perceptual audio coding, traditional sound quality measures such as signal-to-noise ratio (SNR) or signal frequency bandwidth are next

to useless. In practice, *listening tests* are the only reliable method to compare the quality of perceptual audio coding algorithms.

2.2 The MP3 standard

The ISO/IEC 11172-3 (MPEG-1 audio) standard [2] describes a sound format with one or two sound channels sampled at 32 kHz, 44.1 kHz or 48 kHz, encoded at 32 kbit/s up to 320 kbit/s.

The standard describes layer I, II and III. They offer increasing compression ratios, but also increasing complexity in terms of processing requirements.

Layer III is commonly referred to as "MP3" from the file extension it uses and it has become extremely popular due to its high quality at low bit rates.

With the MP3 format, a typical piece of music can be compressed down to approximately 1 MB/minute and still sound virtually indistinguishable from the 10 MB/minute original.

The following sections briefly describe the workings of an MP3 encoder and decoder. There is also an overview of the bitstream.

2.2.1 Encoder

A block diagram of an MP3 encoder is shown in figure 2.2 and the encoding procedure is explained briefly below. For more details, the interested reader is referred to the MP3 standard [2].

The PCM input is divided into chunks of 576 samples called *granules*. For two-channel inputs, a sample represents two values. In this case, each granule will contain information about two channels, and the following steps will be repeated for the second channel.

The samples are fed through a polyphase filter bank that splits the 576 samples into 32 subbands with 18 samples in each subband.

If a granule is initially silent but contains a sharp attack (a sudden loud sound), the masking thresholds might be improper for the silent part of the granule. This results in a brief burst of potentially audible noise

before the attack. This phenomenon is called *pre-echo*. The amount of pre-echo is reduced by using three *short* time windows with six samples per subband to increase the local time resolution. Three modified discrete cosine transforms, MDCTs, are applied on the resulting window values.

Otherwise, if a granule does not contain a sharp attack, one *long* time window with 18 samples per subband is used and an MDCT is applied on the samples.

The combined output of all subbands now form either 576 frequency samples or three time windows with 192 frequency samples. In the latter case, frequency resolution has been traded for time resolution.

Two granules make up one *frame* in which the two granules share sample storage space and some decoding information. The encoder runs a distortion control loop where it iteratively tries to find the best quantization settings for the two granules so that both the psychoacoustic model is satisfied and the bit rate requirement is met. The sample values are Huffman coded to reduce their space requirement. This forces the encoder to spend most of its time calculating how many bits different combinations of values will occupy in the bitstream. The Huffman tables are fixed and known by both the encoder and the decoder.

Finally, when the bit rate is met, the frame is assembled. Apart from the encoded sample data, a frame consists of a header and side information such as quantization settings and Huffman table identifiers.

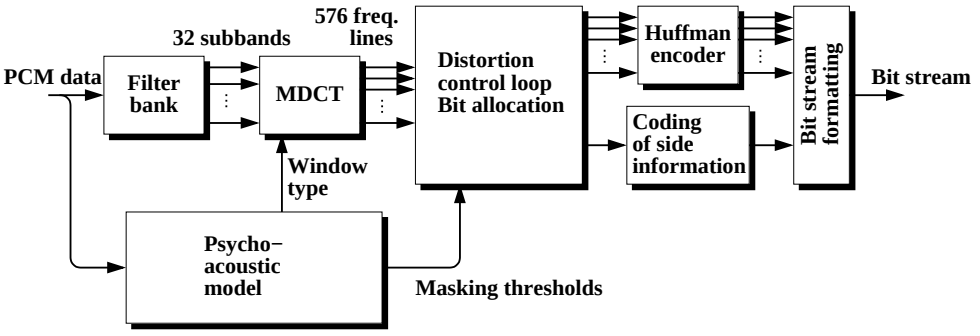


Figure 2.2: Block diagram of an MP3 encoder.

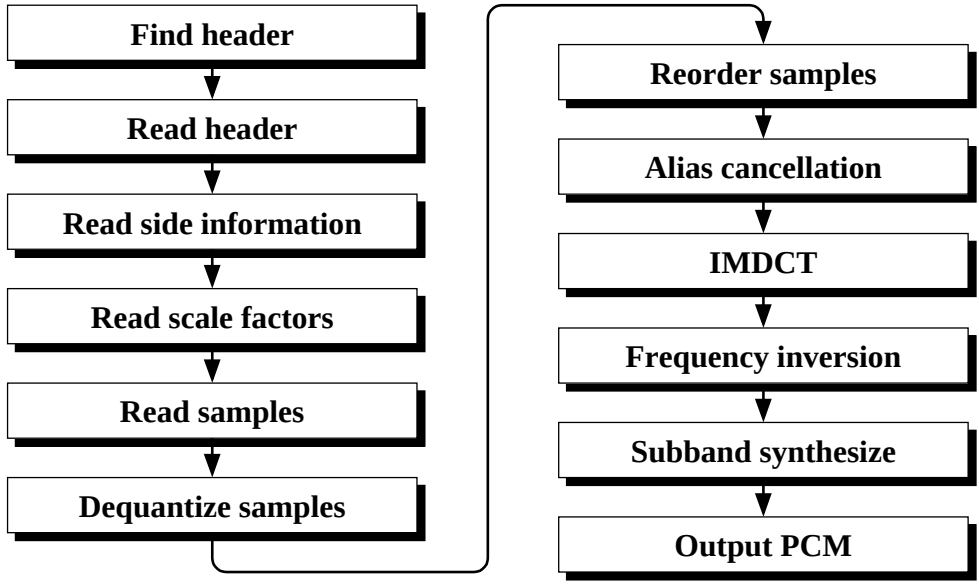


Figure 2.3: Flow chart of an MP3 decoder.

2.2.2 Decoder

The decoder basically applies the inverse transformations to restore the PCM audio stream for playback. All frames are essentially processed in the same way. Figure 2.3 shows a flow chart of the frame decoding process and the steps are described in more detail below.

Find and read header: The first task of the decoder is to locate the synchronization word that marks the beginning of a valid MPEG audio frame.

The synchronization word is part of a header that contains information about the layer number, the sample rate and the channel configuration. These settings are not allowed to change for the duration of the entire bit stream.

The header also contains information about the bit rate which tells the decoder how large the present frame is and thus when to expect the next synchronization word and the next header.

Read side information: The information that is needed by the decoder, apart from the data that eventually will be transformed

back into sample values, is called side information.

There is one side information block for each channel in each granule. This information contains various decoding and dequantization parameters that will be used in the following steps.

Read scale factors: The frequency spectrum is divided into scale factor bands. These bands are determined by the sample rate and they correspond roughly to the critical bands of the human ear.

For each scale factor band, there is a scale factor which will be used later to control the gain during sample dequantization.

Read samples: The 576 Huffman coded sample values are now read and decoded using the Huffman tables indicated by the side information. The encoder may use several different Huffman tables on different sample regions. The various Huffman tables have different number range and/or bit allocation.

The raw sample values are in the range $[-8207 : 8207]$, but the Huffman tables only represent pairs of values that are in the range $[0 : 15]$. In order to code larger values, some tables use the value 15 as an escape code. If the Huffman decoder encounters the escape code, it reads in a table dependent number of bits and adds this value to 15. This number of bits is referred to as *linbits*. The number of linbits varies between 1 and 13. Every non-zero sample is also followed by a sign bit.

Sample dequantization: In this step, the samples from the bitstream are dequantized and scaled to the proper values using the scale factors and the granule gain value. Sample values are raised to the power of 4/3 during the dequantization process.

Reorder samples: Samples in blocks that use the short time window setting (short blocks) must now be reordered in order to be processed by the following steps.

Alias cancellation: The decoder applies alias cancellation to blocks that use the long time window setting (long blocks) to compensate for the frequency overlap of the subband filter bank.

IMDCT: Each subband is now transformed back into the time domain. For long blocks, a 36-point IMDCT calculates the 36 output samples

directly. For short blocks, the output from three 12-point IMDCTs are combined into 36 output samples.

The first 18 output samples are added to the stored overlap values from the previous granule. These values are the new output values. The last 18 output samples are stored for overlap with the next granule.

Frequency inversion: Every second sample in every second subband is now multiplied by -1 to correct for the frequency inversion of the subband filter bank.

Subband synthesize: Finally, the 32 subbands are combined into time domain samples that cover the whole frequency spectrum. One sample is taken from each subband and transformed using a transform similar to DCT. The result is written to the low end of a large array after room has been made by shifting its previous contents towards higher indices. The PCM samples are then calculated by means of a windowing operation on the array.

According to [3], the quality of a decoder is tested by decoding a special reference bitstream called *compl.bit* [4] and comparing the result to a supplied reference signal. Assuming that the output PCM samples are in the range $[-1 : 1]$, to be qualified as a *full precision* decoder, the root mean square, RMS, of the difference signal must not be larger than $2^{-15}/\sqrt{12}$. In addition, the largest absolute difference of a single sample must not be larger than 2^{-14}.

Otherwise, if the RMS of the difference is less than 2^{-11}, the decoder is qualified as a *limited accuracy* decoder, regardless of the largest absolute difference of a single sample. If the output from the decoder does not fulfill any of the requirements, it is *not compliant*.

2.2.3 The bitstream

The output of an MP3 encoder is a self-contained bitstream that contains all information required by an MP3 decoder to restore the original sound. The bit rate of the bitstream can be fixed and known in advance which leads to fixed transmission rates and lower latency due to reduced need for buffering at the receiver. This is of less importance in systems where the entire bitstream is available for random access. For these applications,

the bit rate may be variable to improve audio quality or reduce the size of the bitstream.

There is a header with a synchronization word at the beginning of each frame. The header identifies the bitstream as an MPEG audio layer III stream and gives details about how to interpret the rest of the frame. The header contains settings such as channel configuration and the sample rate.

The headers are found at regular intervals in the bitstream, unless the stream was encoded with a variable bit rate. In any case, the bit rate indication given in each header is enough to know the position of the next header.

It is possible to include almost arbitrary custom data in the bitstream. Every compliant decoder will simply skip this extra data during the synchronization phase as long as the data does not contain anything that looks like a header. To reduce the possibility of decoding false headers, there is an option to protect the header with a CRC checksum.

Immediately following the header is the *audio data* part which contains information needed during sample decoding and dequantization.

When the decoder has read the audio data, it must read the *main data* part which contains scale factors and the actual sample data. A problem is that the main data part does not necessarily begin after the audio data part. Layer III allows a frame to borrow data space from the previous frame. This feature allows the encoder to build up a *bit reservoir* that can be used for reducing the pressure on complex frames.

The audio data part includes a pointer to where the main data part begins. See figure 2.4 for an example. The pointer can only refer backwards in the stream, and the range is limited. In extreme cases, the previous main data must be padded until current main data is in range for the main data pointer.

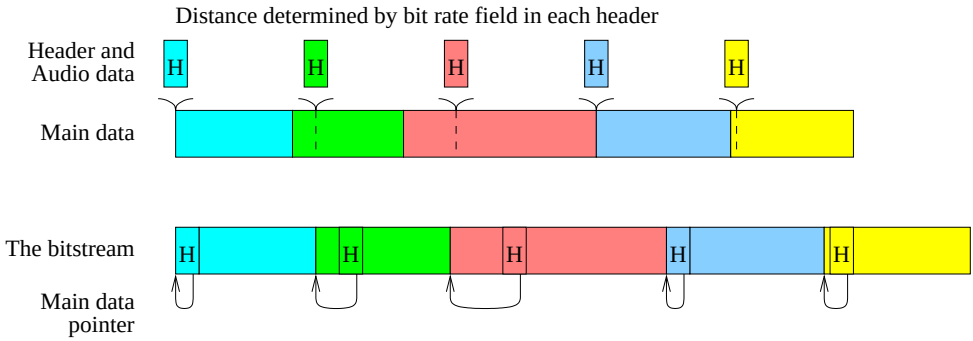


Figure 2.4: The organization of the bitstream. Top: The header and audio data part and the main data part are separated. Bottom: The corresponding bit stream.

3

Floating point format

This chapter describes the tests that were conducted to determine if it was feasible to implement an MP3 decoder using low precision floating point arithmetics.

3.1 Precision

The reference MP3 decoder [5] was modified to use custom wrapper functions around all relevant arithmetic operations. The width of the mantissa and exponent could be changed from the command line. Furthermore, the effects of having two different floating point formats was investigated. This made it possible to study an architecture where the floating point registers are wider than the floating point values stored in memory. Since memory is expensive, both in terms of chip area and power consumption, it was deemed desirable to have an architecture where the intermediate values stored in RAM would be no more than 16 bits wide.

The floating point format used by the wrapper uses an explicit "1." and it does not have any gradual underflow. An overflow was considered a fatal error resulting in a program abort.

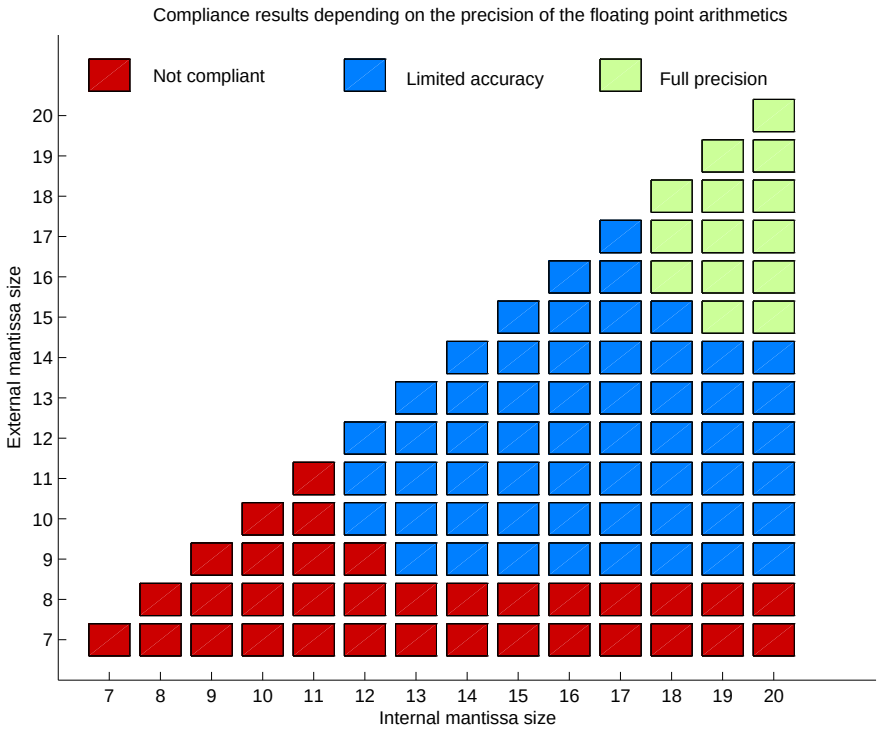


Figure 3.1: A comparison of the compliance result while having different sizes of the internal (register) and external (memory) mantissa.

3.2 Observations

Preliminary tests showed that the exponent had to be at least 6 bits in order to accommodate the dynamic range needed by the algorithms in the reference decoder.

However, experiments on various bitstreams showed that only the sample dequantization used values larger than 2^4. These intermediate values were never stored in memory. Experiments indicated that at least 5 bits need to be stored in memory to get reasonable accuracy. The range of the exponent when stored in memory was chosen as $[4 : -27]$.

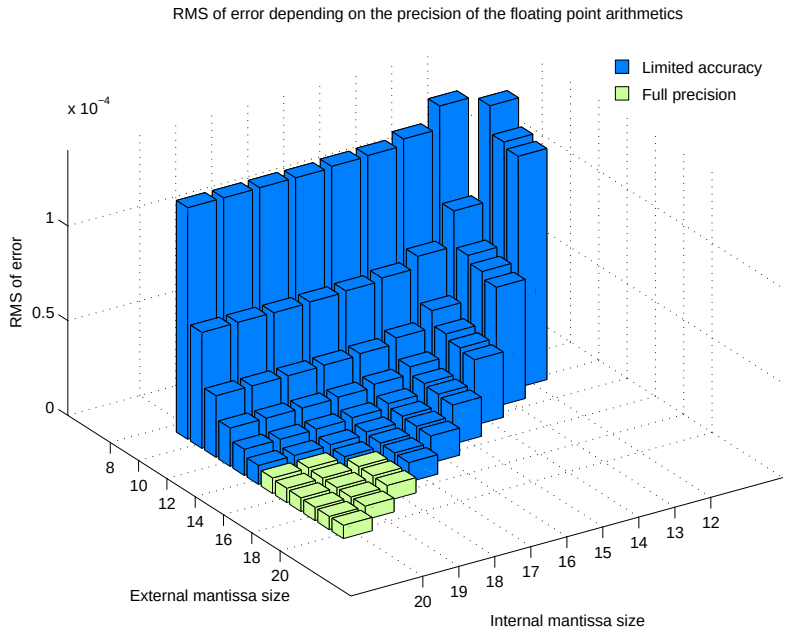


Figure 3.2: A comparison of the RMS error while having different sizes of the internal (register) and external (memory) mantissa. Only configurations conforming to ISO/IEC 11172-3 compliance requirements are present in this figure.

It was subsequently found that it is possible to construct synthetic bitstreams that would cause an overflow to occur. These bitstreams contained as many large values as possible and the global gain was set to the maximum level allowed by the bitstream.

Saturation on overflow would be a possible solution to this problem. Further experiments are needed to determine if this occurs in normal MP3 bitstreams.

Various mantissa configurations were investigated in order to determine a suitable configuration. Figure 3.1 shows the various mantissa size configurations and their corresponding compliance levels. Figure 3.2 shows the RMS error of the compliant decoders. As a comparison, a decoder called *mpg123*, which uses double precision floating point arithmetics, decodes *compl.bit* with an RMS error of $1.3 \cdot 10^{-6}$.

3.3 Optimized algorithms

A profiling of the reference decoder showed that the DCT in the subband synthesis and the inverse MDCT was responsible for a major part of the floating point operations.

The DCT was reduced from a 64-point DCT to a 32-point DCT [6] and was replaced with a version based upon Lee's algorithm [7]. This improved performance from 2048 multiplications and additions to 80 multiplications and 209 additions.

The IMDCT was implemented with an optimized version [8]. This improved performance from the original 648 multiplications and additions down to 43 multiplications and 115 additions.

Using the faster algorithms did not change the RMS noticeably. The algorithms are available as *matlab* programs in appendix D and in appendix E.

3.4 Required operations

An analysis of the reference decoder was conducted to determine the required floating point operations. Only floating point addition, subtraction and multiplication are required to implement an MP3 decoder.

The trigonometric functions can be replaced by tables of a reasonable size and all divisions can be rewritten by using either tables or multiplications.

Calculating $x^{4/3}$ as needed by the sample dequantization could be done by using Newton-Raphson iteration to estimate $x^{-1/3}$, then calculating $(x^{-1/3} \cdot x)^2 = x^{4/3}$. No divisions are necessary in that case. This method was subsequently found to be ineffective and another solution was implemented. The new approach is described in section 6.1.2.

4

Hardware architecture

This chapter describes the hardware architecture of the CPU.

4.1 Overview

The CPU is essentially a RISC processor with signal processing extensions. It features, among other things, separate program, data and constant memories, fixed point and floating point arithmetics, 16 general purpose registers and accelerated sequential bit-level access to the data memory for bitstream decoding. Figure 4.1 gives an overview of the system.

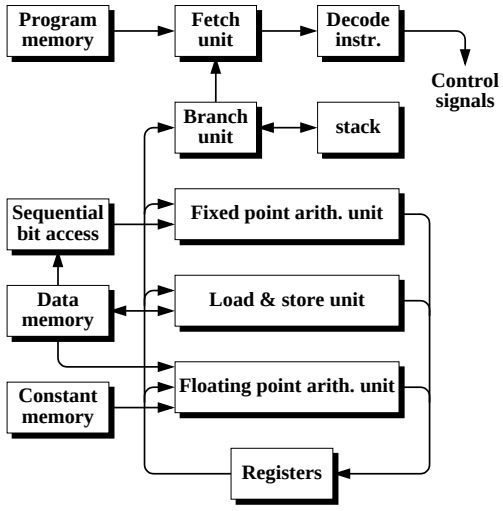


Figure 4.1: Overview of the hardware architecture.

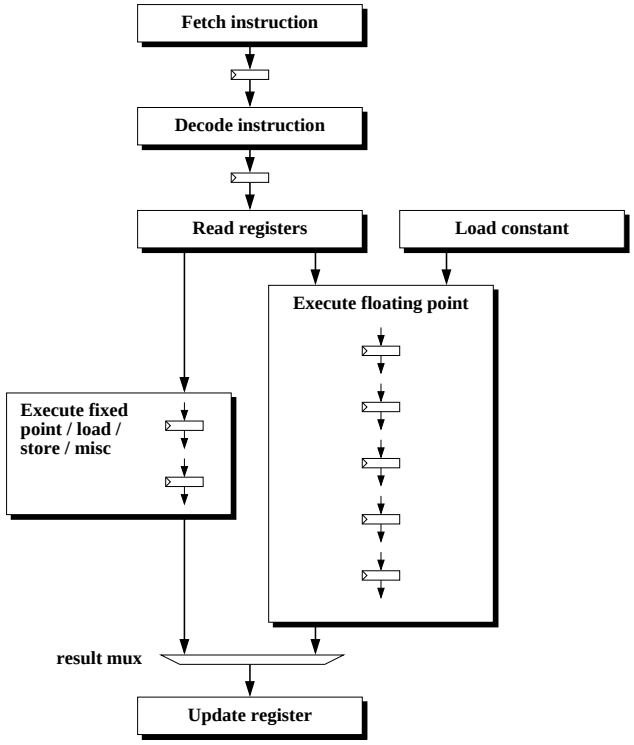


Figure 4.2: The execution pipeline. There are two pipeline registers inside the left execution box and five pipeline registers inside the right execution box.

The primary design strategy was to keep the hardware as simple as possible by moving as much complexity as possible to the software. Our experience from previous projects is that it is usually easier to verify and adapt software to hardware limitations than vice versa.

The intention was to pipeline the hardware for speed under the presumption that if it can be clocked at a very high frequency, it can also run with a low core voltage (at a lower clock frequency) to get low power consumption. Unfortunately, the relatively high number of pipeline steps require non-trivial instruction scheduling in order to obtain maximum performance. The pipeline diagram is shown in figure 4.2. More details are shown in figure 4.4, 4.5 and 4.6. These figure are discussed in sections 4.3, 4.4 and 4.5, respectively.

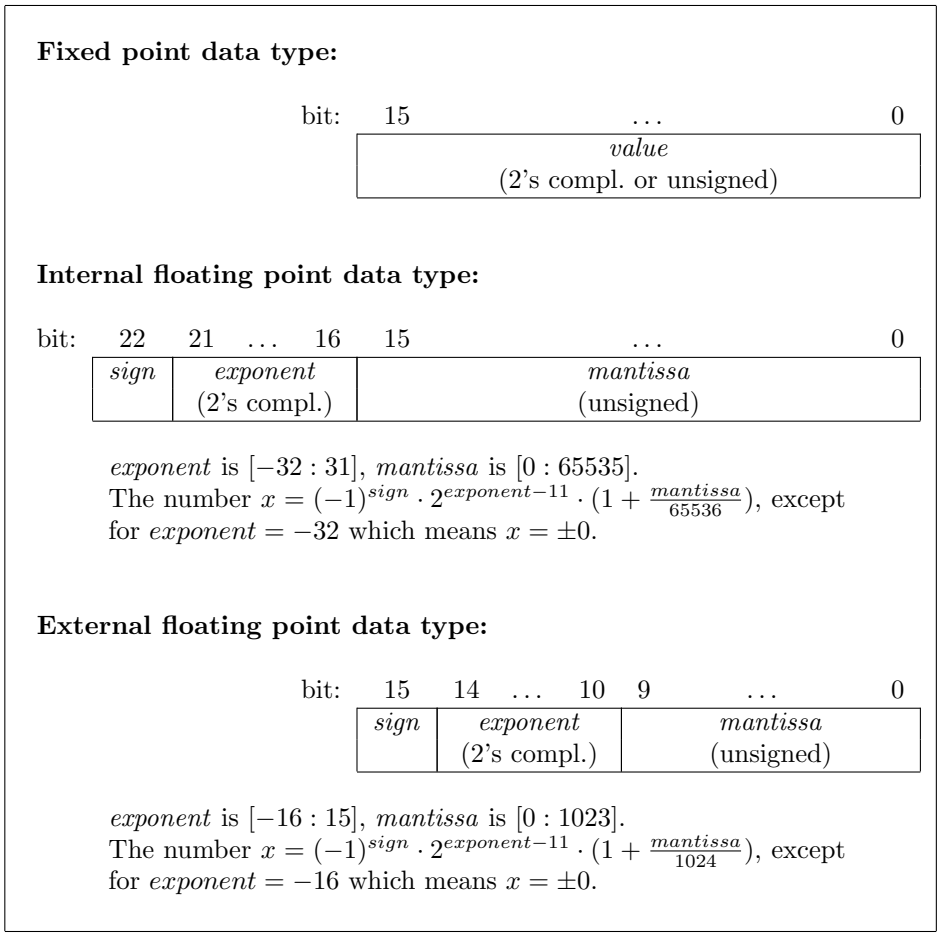


Figure 4.3: The main data types.

4.2 General purpose registers

There are 16 general purpose registers named `r0` through `r15`. This number of registers was chosen in order to have a reasonably small register file to reduce the width of the instruction words while still allowing certain algorithms to keep all their temporary variables in registers.

The registers are 23 bits wide and the data type associated with these bits depends on the executed instruction. Some instructions use the 16 least significant bits of the register contents as a 16-bit fixed point number, typically an integer or a memory pointer. In this case, the seven most significant bits are ignored on reads and unaffected by writes. Other

instructions use the register contents as a 23-bit floating point number. Figure 4.3 gives a bit level description of the data types.

A third data type, a 16-bit floating point type, is used for storing floating point values in memory. There are no instructions for manipulating data with this type, except type conversion instructions.

The register file has two read ports and one write port. All arithmetic instructions that use registers can use any two registers as source operands and any register as destination for the result.

4.3 Special purpose registers

Apart from the general purpose registers, there are several special purpose registers. They have predetermined functions in the processor.

The contents of a special register can be copied to and from a general purpose register. It may also be implicitly updated as the result of another operation. An example of this could be a special register that is used as a memory pointer. When the pointer has been dereferenced, it is implicitly incremented to point to the next memory value.

The instruction encoding allows for 16 special purpose registers, sr0 through sr15, but only sr0–sr5 and sr12–sr15 are implemented. The result of accessing an unimplemented special register is undefined.

sr0 is used in the bit reader for holding up to 16 bits before they are read by the CPU. When all bits have been read, this register is immediately updated with the next word from a prefetch register. A few clock cycles later, a new word is read from the data memory into the prefetch register ready to be used the next time sr0 is empty.

sr1 is a 4-bit counter that counts how many bits there would be left in sr0 after the next read. A read from sr0 when this register is zero triggers the reload hardware.

sr2 is used as a data memory pointer by the bit reader and the fmac and fmaci instructions. It is automatically incremented by the bit reader and by fmaci.

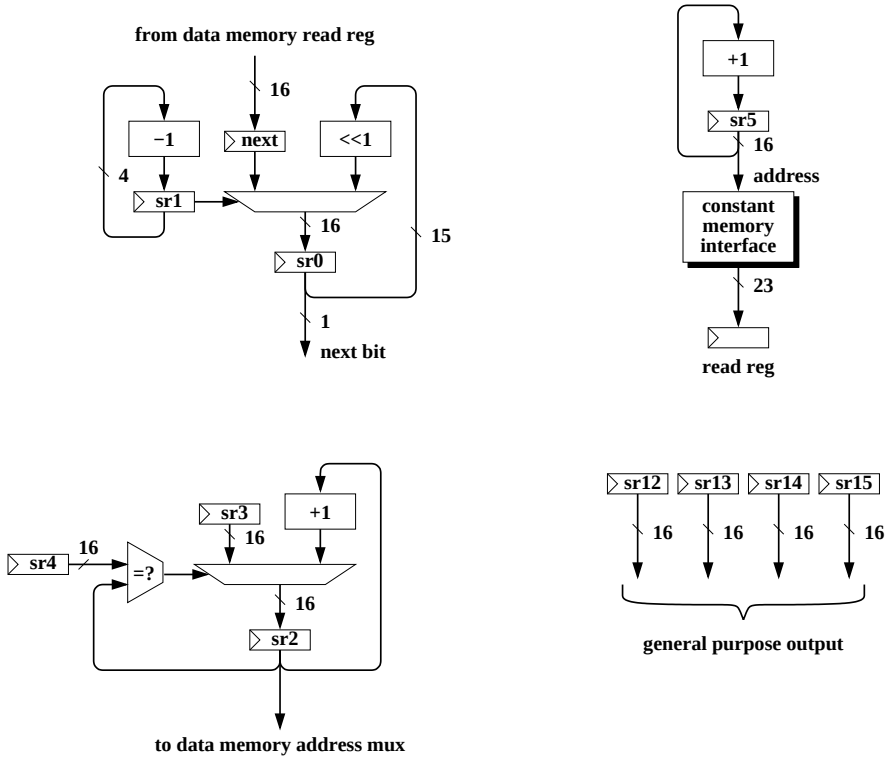


Figure 4.4: The special registers.

sr3 contains the restart value for sr2. When sr2 is about to be incremented past the contents of sr4, it is instead set to the contents of sr3. This register is write-only.

sr4 contains the end value for sr2. The registers sr3 and sr4 are typically used for marking the beginning and the end of a circular buffer. This register is write-only.

sr5 contains the constant memory pointer that is used by fmulc, fmac and fmaci. It is automatically incremented each memory read. This register is write-only.

sr12–**sr15** are 64 general purpose inputs (on reads) and 64 general purpose outputs (on writes). They are typically used for communicating with peripherals such as a sample FIFO and a bitstream FIFO.

Figure 4.4 gives a graphical representation of the special registers.

4.4 Fixed point data path

The fixed point data path is shown in figure 4.5. All units operate on 16-bit data, except the shifter that implements the seth, ltoh and htol instructions.

The arithmetic unit implements simple two's complement arithmetics. Overflow is handled by wrapping and the carry out is lost.

The origin of the next bit signal is shown in figure 4.4. The branch taken and new pc signals go to the instruction fetch unit.

4.5 Floating point data path

The floating point data path is shown in figure 4.6 and it consists of one floating point add/subtract pipeline and one floating point multiply pipeline. The add pipeline is also used for the fpack and fint instructions since they involve rounding which is similar to adding. In the multiplication pipeline, the mantissa multiplication step is divided into three steps to reduce the combinational delay.

The data memory read register is found in figure 4.5 and the constant memory read register is found in figure 4.4.

If overflow occurs during an operation, it is not handled and the result is undefined. Underflow is handled by setting the result to zero.

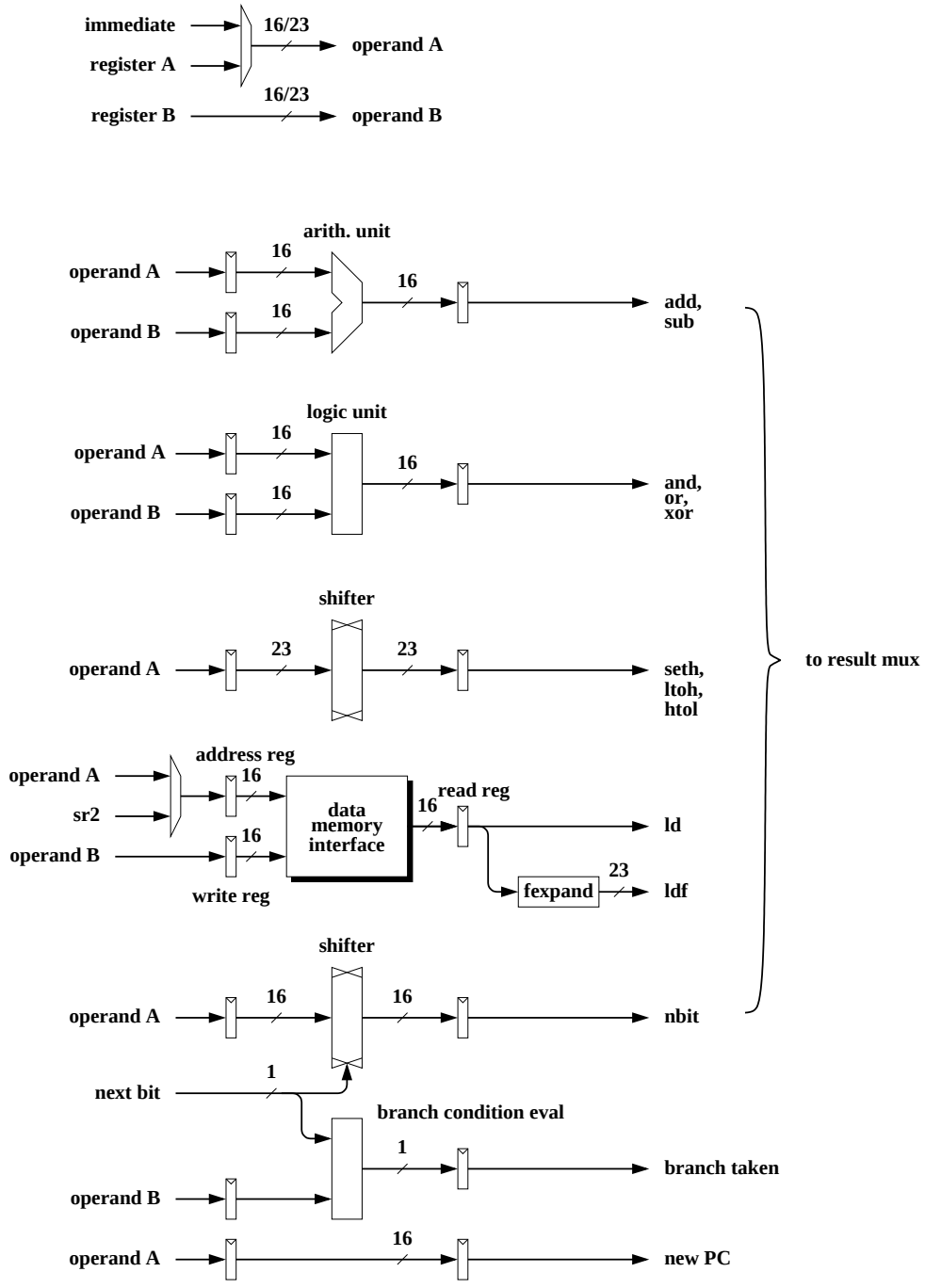


Figure 4.5: The fixed point data path. At the top is the operand mux which selects operand A. Operand B is always taken from the register file.

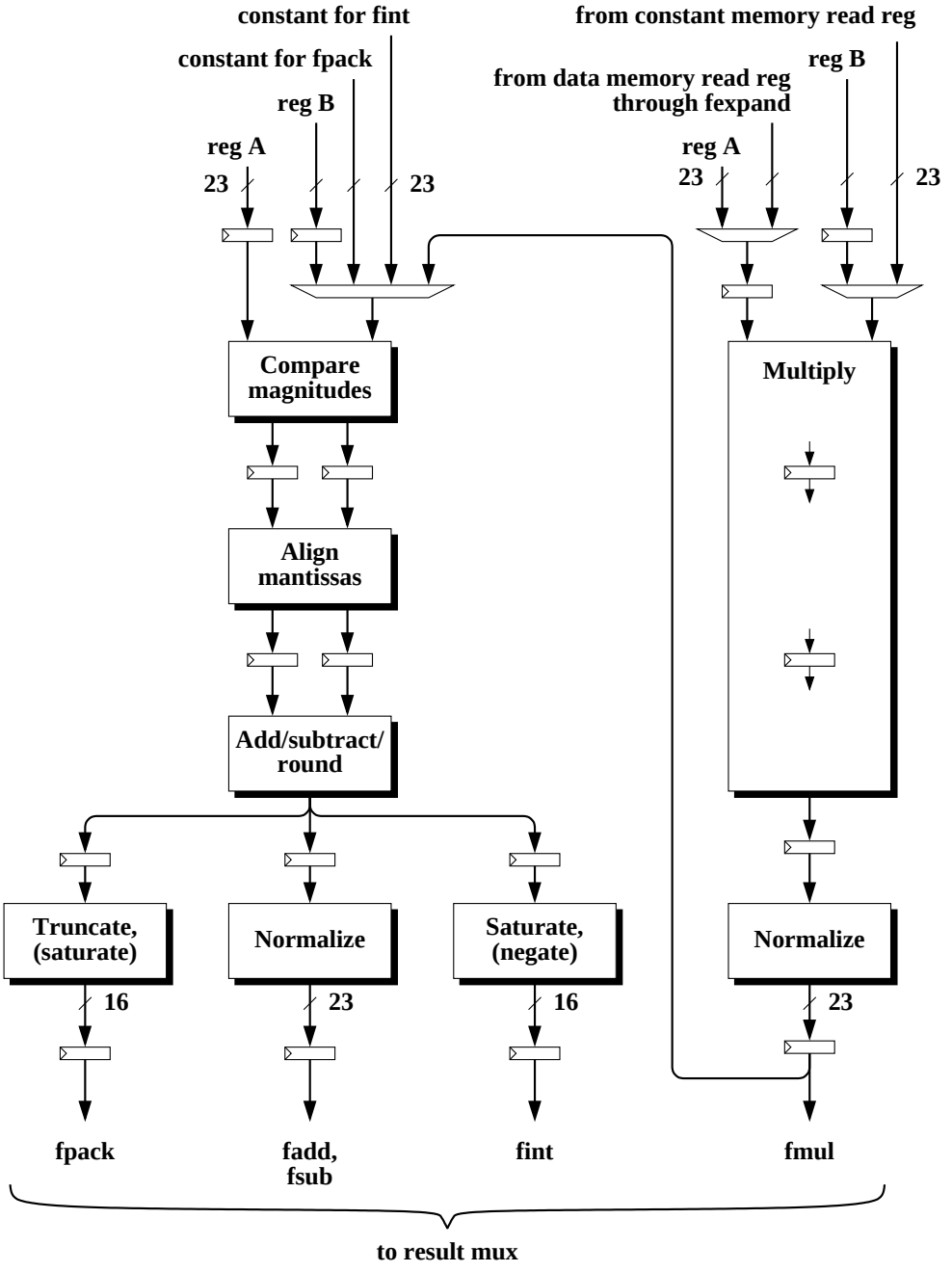


Figure 4.6: The floating point data path. The multiplier is pipelined with two pipeline registers.

4.6 Memory interfaces

There are three different memories in the system: the program memory, the data memory and the floating point constant memory.

4.6.1 Program memory

Under normal conditions, the CPU fetches a new instruction from the program memory every clock cycle. The program memory is 24 bits wide and up to 64K words deep. It can only be accessed by the instruction fetch unit, therefore it cannot easily be used for storing tables of constant data for the program, although small tables can be implemented using computed jumps.

4.6.2 Data memory

The data memory is used for storing run-time data, its contents are undefined after a system reset. It is 16 bits wide and is used for storing data in the fixed point format or the external floating point format. The CPU can address up to 64K words of data memory.

4.6.3 Constant memory

The constant memory is a 23-bit wide ROM used for storing floating point constants such as window coefficients. It is addressed by a dedicated pointer register and the read constant is always fed to the multiplier pipeline. Since it has a dedicated pointer register, there is no hard limit on the size of the constant memory.

4.7 Instruction set

The RISC-like instruction set is very limited. For example, there are no shift instructions and a very limited set of conditional branches. There are some task-specific instructions, however. The instruction set is discussed in detail in appendix A and the instruction encoding is given in appendix B.

5

Tools

A set of tools was implemented to be used during the implementation and verification of the CPU and the MP3 decoder. All tools were implemented in GNU C.

5.1 Instruction set simulator

In order to test run program code, a clock cycle and pipeline accurate CPU simulator was implemented. It also simulates the memories and other devices connected to the CPU. The simulator is non-interactive, it only displays a clock cycle counter and MIPS statistics when it runs.

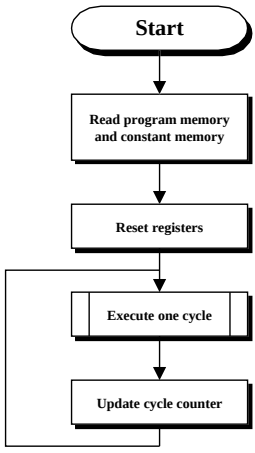


Figure 5.1: Flowchart of the instruction set simulator.

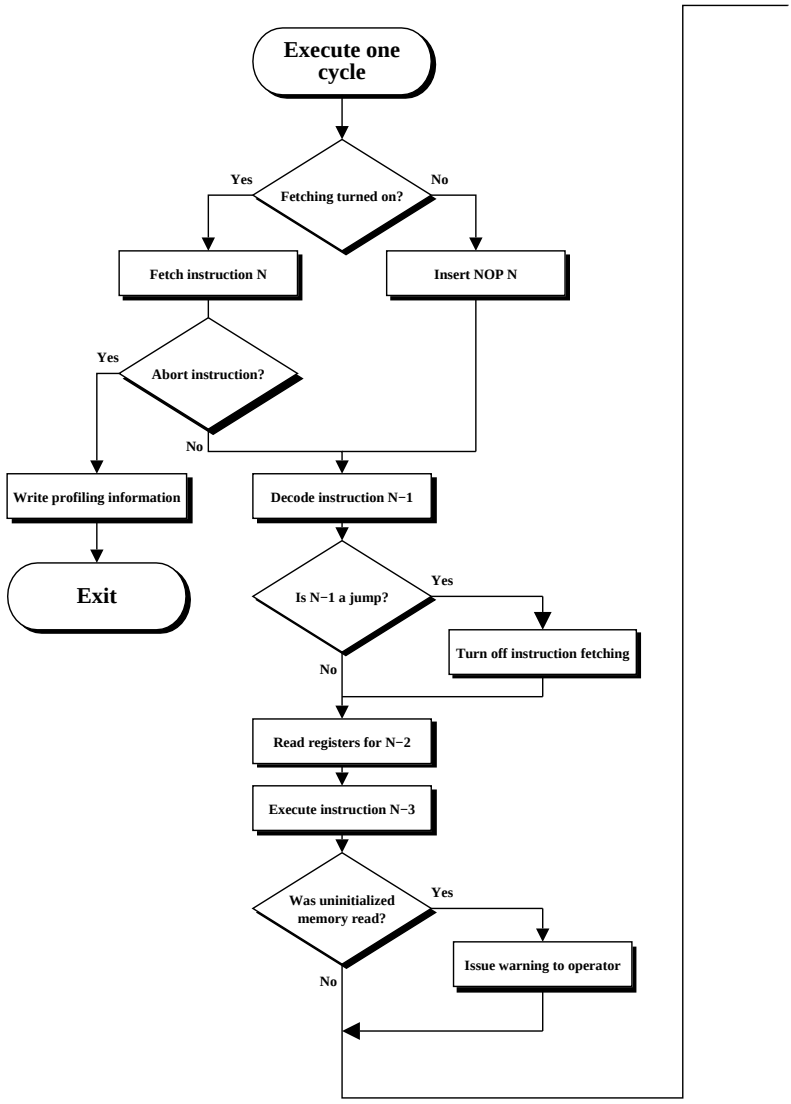


Figure 5.2: Flowchart of how the instruction set simulator executes one cycle.

The simulator begins by reading the input file that consists of a hexadecimal dump of the contents of the program memory and the constant memory. The file also contains a symbol table. The simulation stops when it encounters an abort instruction.

During execution, the simulator gathers execution profiling information.

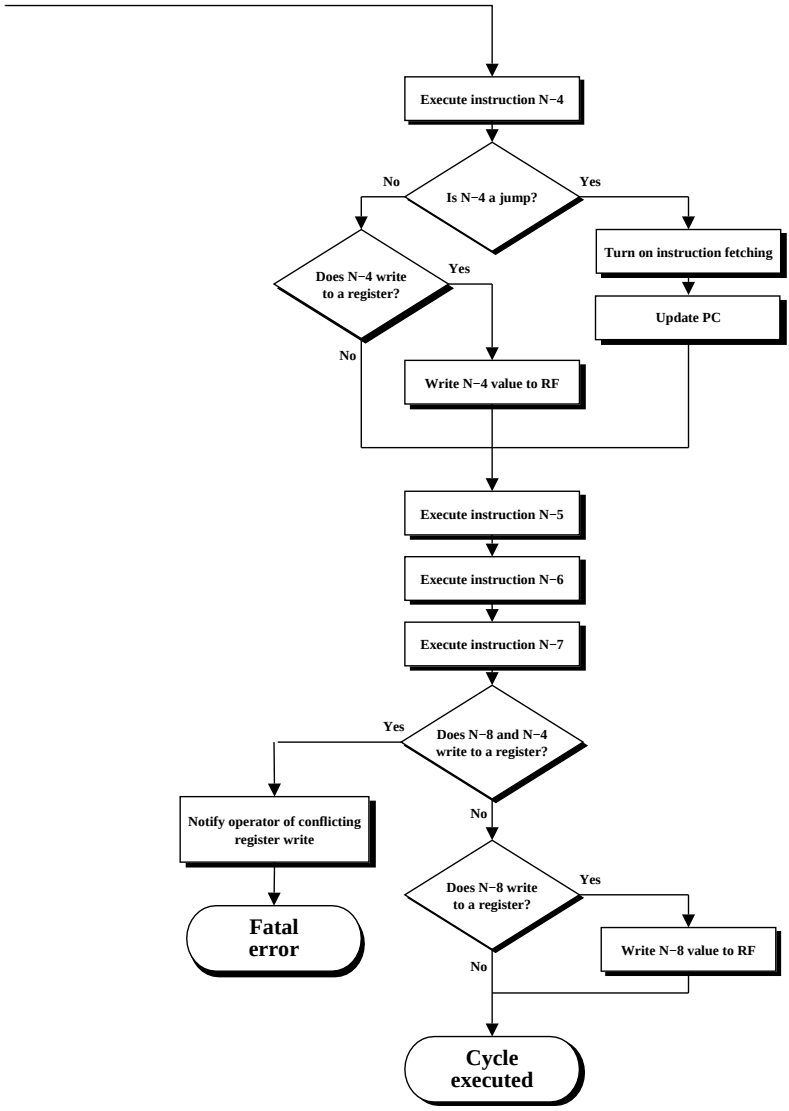


Figure 5.2: Flowchart of how the instruction set simulator executes one cycle. (contd.)

Every program memory address has a fetch count associated with it and every time an instruction is fetched, the corresponding fetch count for that address is incremented. When the simulation is finished, a detailed execution profiling report is generated from the fetch counts with help of the symbol table. This report contains information about how much

CPU time is spent in every function, instruction usage statistics and other interesting data.

There are no real debugging facilities implemented in the simulator except the option to produce a memory access log. This file contains one line for every data memory read or write issued by the CPU. Each line contains the address and data in question and the corresponding variable name from the symbol table. The memory log file typically grows *very large* very quickly and all the disk accesses slows down the simulator considerably.

5.2 Assembler

The MP3 decoder was implemented entirely in assembly language and an assembler was implemented to convert the assembly language into the corresponding executable machine code.

Each function of the decoder is stored in a separate file. A top level file, which is processed by *cpp* (the C language preprocessor), includes all functions and data declarations into one huge file which is then read by the assembler. This provided support for macros and conditional assembling without any extra effort.

The assembler supports local labels to reduce the risk of name clashing and to improve the profiling output from the instruction set simulator. All instructions are supported along with several variants and convenience macros.

The assembler output contains machine code for the program memory, the contents of the constant memory and the symbol table. The use of *cpp* eliminates the need for a linker.

5.3 Huffman table compiler

The core of the Huffman decoding stage of the MP3 decoder was automatically generated by a special Huffman table compiler that was implemented for this purpose. The compiler was run on the Huffman tables file from the reference decoder (*huffdec*) and the output was then copied into the appropriate function in the decoder.

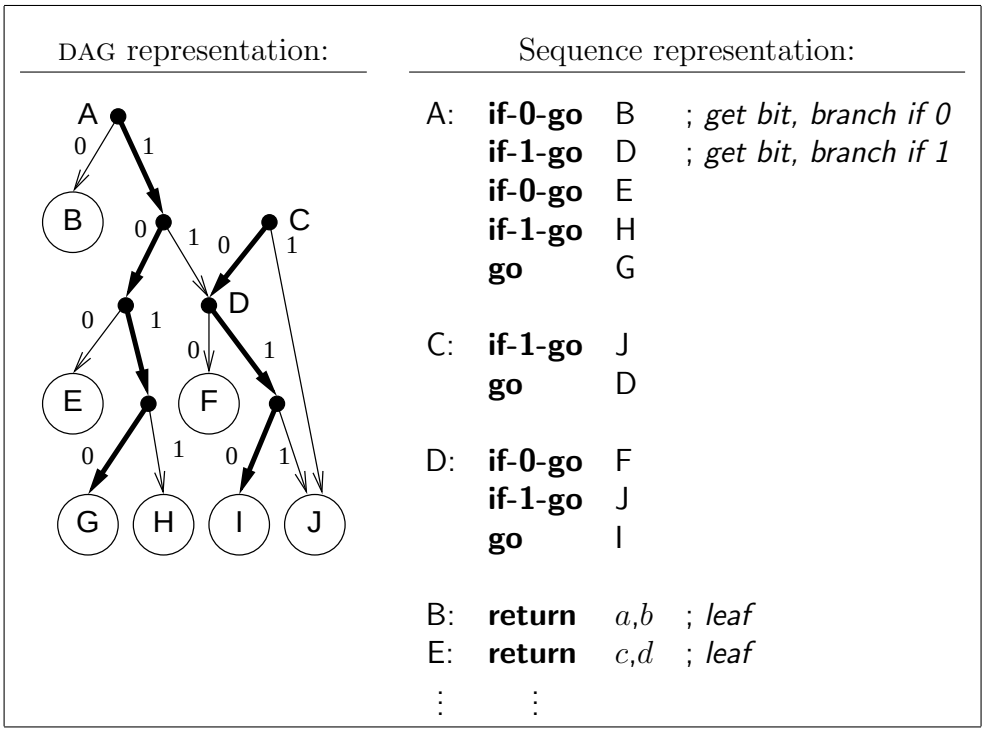


Figure 5.3: Stage 3 (sequence extraction) of the Huffman table compiler.

The algorithm used by the Huffman table compiler is outlined below:

1. The input file is read and a binary tree representation of all Huffman trees is formed.

2. All identical nodes and leaves are merged. The internal representation is now a directed acyclic graph.

3. An intermediate representation similar to code is formed. It consists of branch instructions and leaf instructions. The fall-through paths of the branches are chosen to obtain the longest possible sequences of instructions, see figure 5.3.

4. Real code for the CPU is now generated by interleaving branch sequences of matching length, see figure 5.4. If there are no sequences with matching lengths, NOP instructions are appended to make an existing sequence longer.

Instruction			Seq.		
A:	**if-0-go**	B	A		
C:	**if-1-go**	J		C	
	if-1-go	D	A		
	go	D		C	
	if-0-go	E	A		
D:	**if-0-go**	F			D
	if-1-go	H	A		
	if-1-go	J			D
	go	G	A		
	go	I			D

Figure 5.4: Three interleaved sequences.

Interleaving makes sense because the conditional bit branch instructions force the CPU to fetch but not execute the following instruction. Interleaving is not strictly necessary, but it reduces the code size since the alternative would be to put a NOP instruction after each conditional branch. This is a significant improvement in the branch intensive Huffman decoder.

5. Peephole optimizations for common instruction sequences and jump optimizations such as replacing a branch with its target are now applied.

   The patterns to look for were found by manually inspecting the input to this stage.

6. Finally, the code is converted to an assembler-readable (human-readable) textual representation and it is saved to disk.

The final size of the decoder core is 2223 instructions which compares well to the 1379 leaves and 1378 inner nodes of the Huffman tables (not counting identical tables). The code size before the peephole optimization stage is 2882 instructions.

The decoder core does not handle *linbits* or sign bits, therefore wrapper functions are needed for each table to make it usable in an MP3 decoder.

6

MP3 decoder implementation

This chapter describes the MP3 decoder that was implemented to run on the previously described CPU.

6.1 Components

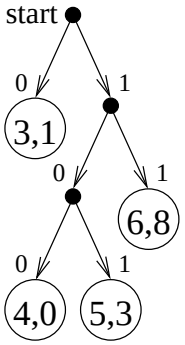
In order to understand the algorithms better, an MP3 decoder was first implemented in C. This decoder was considerably smaller than the reference decoder, partly because only layer III was supported. This decoder served as a basis and reference for the assembly language implementation.

Most of the assembly language implementation was a straight forward translation of the C implementation, but the performance critical parts received more attention. These parts are described in this chapter.

6.1.1 Huffman decoder

The Huffman decoder was implemented as code where each Huffman tree node was implemented as a conditional branch instruction. Table 6.1 contains a simple example that illustrates the technique. The obvious solution to iterate over tables stored explicitly as constant data could not be used since there is no suitable constant memory available. The code solution is also faster, at the expense of memory usage.

A program was written to read the Huffman tables supplied with the reference decoder and compile them into the corresponding program. During the compilation, some optimizations were applied to reduce the



Corresponding code:

```
start:    bnbc   got0     ; branch if bit is 0
          (nop)           ; bit was 1
          bnbs   got11    ; branch if bit is 1
          (nop)           ; bit was 0 (got 10)
          bnbs   got101
          (nop)           ; bit was 0 (got 100)
got100:   set    #4,r0    ; return 4 in r0
          ret
          set    #0,r1    ; return 0 in r1

got0:     set    #3,r0    ; return 3 in r0
          ret
          set    #1,r1    ; return 1 in r1

got101:   set    #5,r0    ; return 5 in r0
          ret
          set    #3,r1    ; return 3 in r1

got11:    set    #6,r0    ; return 6 in r0
          ret
          set    #8,r1    ; return 8 in r1
```

Table 6.1: Principle of Huffman table decoder.

size of the code. The compiler and the optimizations are described further in section 5.3.

6.1.2 Sample dequantization

The performance critical part of the sample dequantization turned out to be calculating $|x|^{4/3} \cdot \text{sign}(x)$ where x is an integer in the range $[-8207 : 8207]$.

The trivial solution is a large table with precalculated values. This solution was implemented for comparison purposes only as it was too memory inefficient to be used in practice. Several other algorithms were tried, they are discussed below. These algorithms are summarized in table 6.2

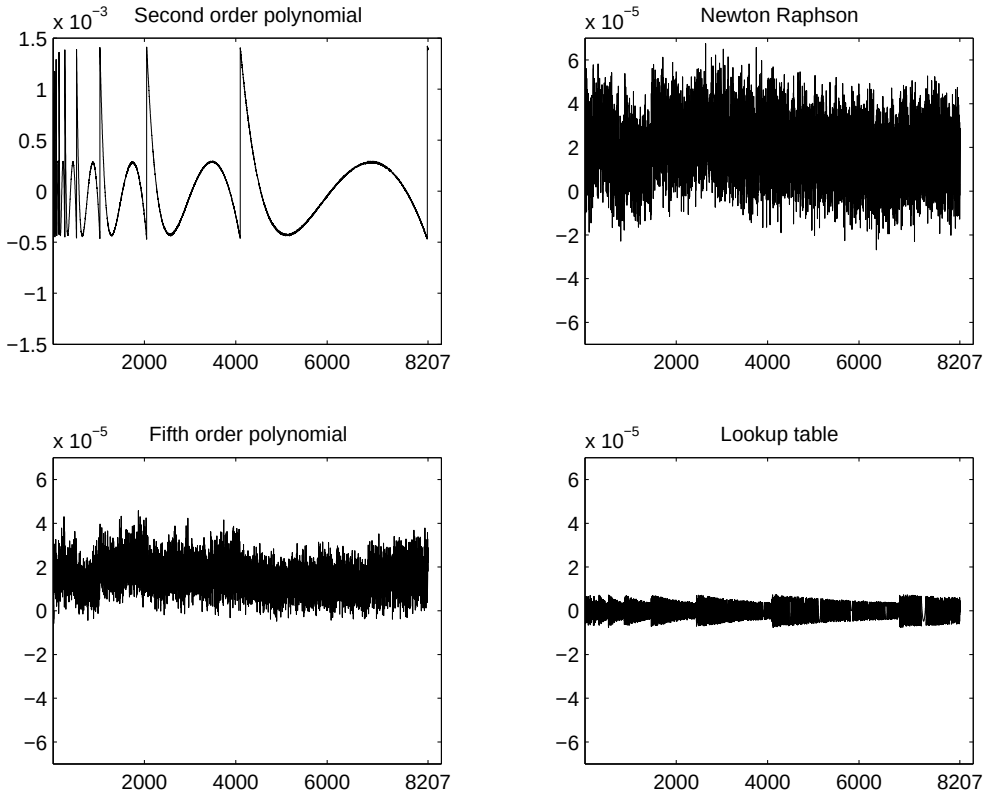


Figure 6.1: The relative error of $x^{4/3}$ for various algorithms.

Algorithm	RMS of error
Lookup table	$3.2 \cdot 10^{-5}$
Newton-Raphson	$3.2 \cdot 10^{-5}$
Polynomial (2nd order)	$4.0 \cdot 10^{-5}$
Polynomial (5th order)	$3.2 \cdot 10^{-5}$

Table 6.2: RMS error while decoding *compl.bit* with different algorithms for $x^{4/3}$.

and figures 6.1 and 6.2. They show that numerical algorithms (that do not rely on big tables) can be implemented with roughly the same precision as a table based approach.

The initial implementation used Newton-Raphson iteration. In order to avoid the need for a floating point division, Newton-Raphson was used for estimating $x^{-1/3}$ which then was used for calculating $x^{4/3} = (x^{-1/3} \cdot x)^2$.

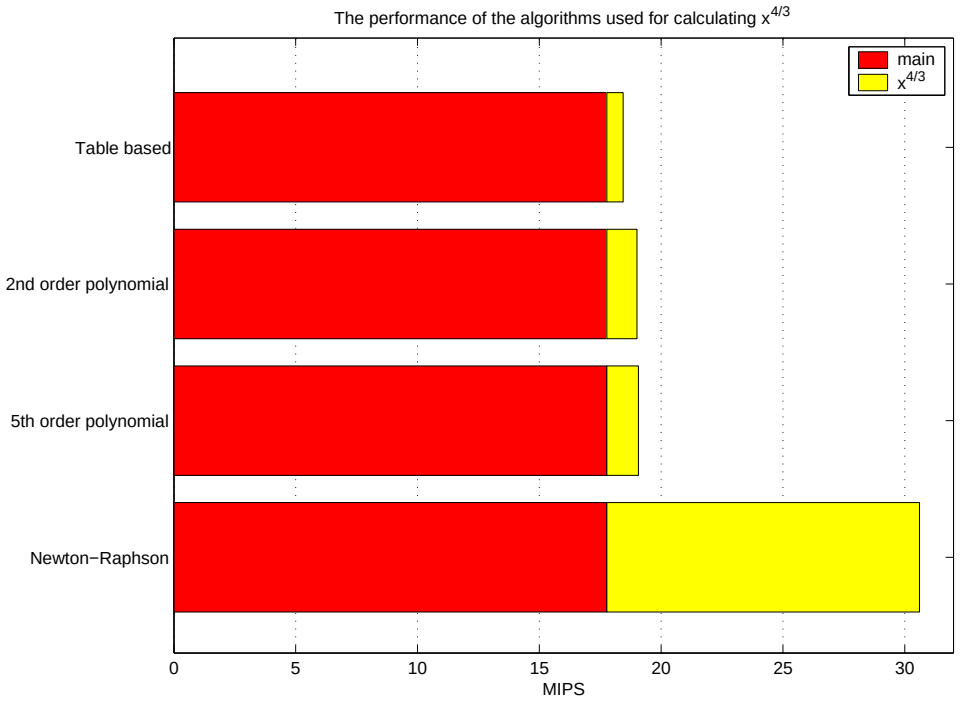


Figure 6.2: A comparison of the performance of different algorithms for calculating $x^{4/3}$. The performance was measured on a 48 kHz bitstream designed to make the sample dequantization as hard as possible for the decoder. The comparison is not completely fair because the Newton-Raphson algorithm lacks some optimizations for common special cases.

While the accuracy of this algorithm is good as long as enough iterations were used, the performance is very poor. The algorithm operates on a *pair of samples* to improve performance by parallel computations, as does the other three algorithms.

The fastest practical algorithm used approximation by means of second order polynomials. Different polynomials were used depending on the exponent. The coefficients were selected using the least squares method. The relative error was considerably worse than the other algorithms and the RMS of the error signal was noticeably worse with this algorithm.

Finally, an algorithm that uses a fifth order polynomial approximation was implemented. This algorithm is slightly slower than the second order

polynomial but the precision was once again almost as good as the table based approach. This algorithm was used in the final decoder and a *matlab* version can be found in appendix C.

6.1.3 IMDCT

The inverse modified discrete cosine transform, IMDCT, used in MP3 decoding is shown in equation 6.1. This 36-point IMDCT is valid for long blocks. Short blocks use a 12-point IMDCT.

$$x_i = \sum_{k=0}^{17} X_k \cos\left(\frac{\pi}{72} \cdot (2 \cdot i + 19) \cdot (2 \cdot k + 1)\right) \qquad i \in 0, \ldots, 35 \quad (6.1)$$

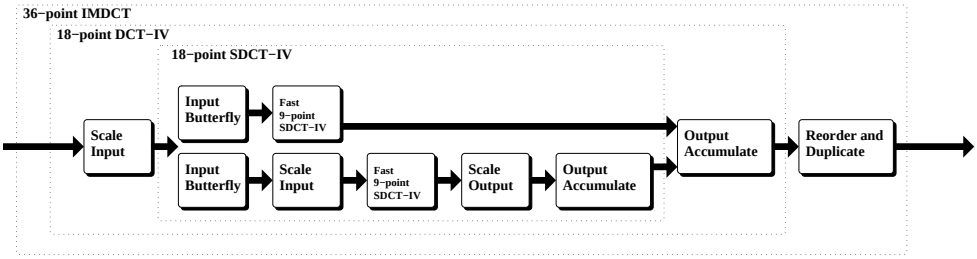


Figure 6.3: Overview of the IMDCT optimization.

The 36-point IMDCT was implemented with an algorithm proposed by Szu-Wei Lee [8]. The algorithm divides the IMDCT into two 9-point scaled DCTs as shown in figure 6.3. The 9-point SDCT-II was implemented using only 8 multiplications and 36 additions. A *matlab* implementation of this algorithm can be found in appendix D.

The algorithm relies on an accumulation stage that is troublesome on architectures with long pipelines. The windowing operation was done in parallel with the accumulation, thereby making use of otherwise empty pipeline slots.

There are other ways to implement the IMDCT. An algorithm proposed by Britanak and Rao [9] does not depend upon a long accumulation stage. The downside is that this algorithm need more operations. It

would not improve performance unless a change in the instruction set made it possible to optimize the windowing operation.

The 12-point IMDCT that is used in short blocks can be optimized in a similar way by reducing it to two 3-point scaled DCTs. The current implementation does not do this. The 12-point IMDCT is reduced to a 6-point DCT-IV which is calculated by a straight-forward matrix multiplication implemented with the fmac and fmaci instructions.

6.1.4 Subband Synthesis

The subband synthesis consists of two parts, a DCT operation and a windowing operation. The reference decoder uses a 64-point DCT on 32 subband samples as illustrated in equation 6.2. x is fetched from the output of the frequency inversion step of the decoder.

$$X_i = \sum_{k=0}^{31} x_k \cos\left(\left(\frac{\pi \cdot i}{64} + \frac{\pi}{4}\right) \cdot (2k+1)\right) \qquad i \in 0, \ldots, 63 \tag{6.2}$$

A 1024-entry large array V contains the result of the DCT operation. The contents of V is shifted and the result of the DCT operation is inserted (equation 6.3). A total of 16 DCT operations are stored in V.

$$\begin{aligned} V'_i &= V_{i-64} && i \in 1023, \ldots, 64 \\ V'_i &= X_i && i \in 63, \ldots, 0 \end{aligned} \tag{6.3}$$

Finally, the PCM samples are calculated as described by equation 6.4. D contains the coefficients of the synthesis window described in annex B of the MP3 standard.

$$\text{Sample}_j = \sum_{i=0}^{7} V_{128 \cdot i + j} \cdot D_{64 \cdot i + j} + V_{128 \cdot i + j + 96} \cdot D_{64 \cdot i + j + 32} \qquad j \in 0, \ldots, 31 \tag{6.4}$$

It is easy to reduce the 64-point DCT to a 32-point DCT by duplicating some results and changing the sign as appropriate [6]. The resulting DCT is shown in equation 6.5. The relation between X' and X is outlined in equation 6.6.

$$X'_i = \sum_{k=0}^{31} x_k \cos\left(\frac{\pi}{2 \cdot 32} \cdot (2 \cdot k + 1) \cdot i\right) \qquad i \in 0, \ldots, 31 \tag{6.5}$$

$$\begin{array}{lcll} X_i & = & X'_{i+16} & \\ X_{16} & = & 0 & \\ X_{i+17} & = & -X'_{31-i} & i \in 0, \ldots, 15 \\ X_{i+33} & = & -X'_{15-i} & \\ X_{i+48} & = & -X'_i & \end{array} \tag{6.6}$$

The 32-point DCT was implemented using Lee's algorithm [7]. It is possible to divide an N-point DCT into two $N/2$-point DCTs. If N is a power of 2, a DCT can be recursively divided in an optimal way. Figure 6.4 illustrates this. A *matlab* implementation of this algorithm can be found in appendix E.

By investigating the array indices in equation 6.4, it is clear that only half

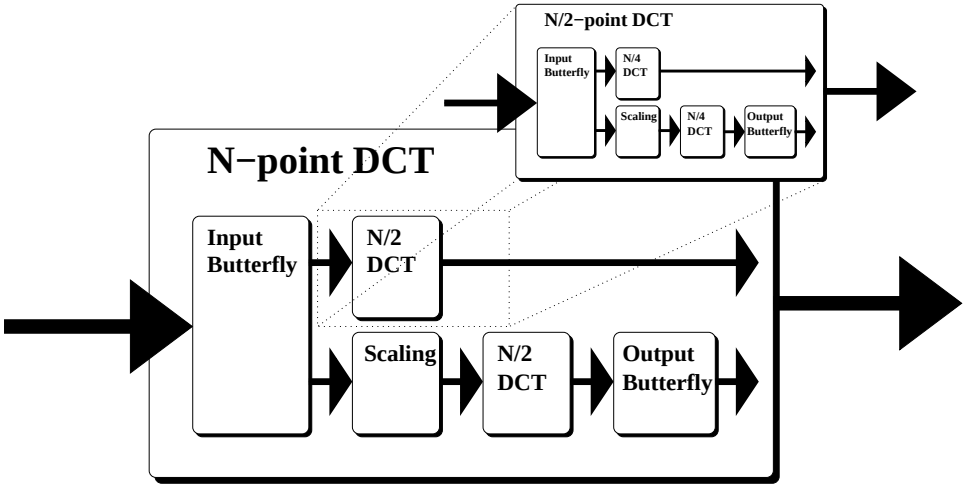


Figure 6.4: How to recursively divide a DCT into smaller DCTs using Lee's algorithm.

of the array is accessed at a time. The addressed elements are shown in equation 6.7. The shift operation in equation 6.3 ensures that all values are used at some point.

$$\begin{matrix} 0,\ldots,31 \\ 128\cdot i+96,\ldots,128\cdot i+159 \qquad i\in 0,\ldots,6 \\ 992,\ldots,1023 \end{matrix} \tag{6.7}$$

V can thus logically be divided into two arrays, V^{odd} and V^{even}. One array that is used every even windowing operation and one array that is used for every odd windowing operation.

Further optimizations can be made by not duplicating values as seen in equation 6.6. V could be reduced to 512 words in this way. Unfortunately one value appear in both the even and odd part of V ($X'_{16} = X_0 = -X_{32}$). This makes it hard to divide V into an even and an odd array of 256 words each. Thus, V^{even} and V^{odd} are 272 words each.

The final layout of V^{even} and V^{odd} is shown in table 6.3. The modified subband synthesis windowing is shown in equation 6.8. The other samples are calculated in a similar manner. This allows the fmac and fmaci instructions to be utilized. (These instructions are explained in appendix A.) The pipeline penalty of the FPU is mitigated by calculating several samples in parallel. The modulo addressing mode of these instructions were utilized to eliminate the copying operation of equation 6.3. W' is the new window. It contains the same values as W but the values are rearranged and some values have been negated in order to do the conversion of equation 6.6.

$$
\begin{aligned}
\text{Sample}_0 &= \sum_{i=0}^{15} V_{i\cdot 4+0}^{even} \cdot W'_{i\cdot 7+0} \\
\text{Sample}_1 &= \sum_{i=0}^{15} V_{i\cdot 4+1}^{even} \cdot W'_{i\cdot 7+1} \\
\text{Sample}_{31} &= \sum_{i=0}^{15} V_{i\cdot 4+1}^{even} \cdot W'_{i\cdot 7+2} \\
\text{Sample}_2 &= \sum_{i=0}^{15} V_{i\cdot 4+2}^{even} \cdot W'_{i\cdot 7+3} \\
\text{Sample}_{30} &= \sum_{i=0}^{15} V_{i\cdot 4+2}^{even} \cdot W'_{i\cdot 7+4} \\
\text{Sample}_3 &= \sum_{i=0}^{15} V_{i\cdot 4+3}^{even} \cdot W'_{i\cdot 7+5} \\
\text{Sample}_{29} &= \sum_{i=0}^{15} V_{i\cdot 4+3}^{even} \cdot W'_{i\cdot 7+6}
\end{aligned} \tag{6.8}
$$

$V^{even}_{0,\ldots,63}$ DCT32 output index	$V^{odd}_{0,\ldots,63}$ DCT32 output index	DCT32 output age
$16,\ldots,19$	$16,\ldots,13$	0
$16,\ldots,13$	$16,\ldots,19$	1
$16,\ldots,19$	$16,\ldots,13$	2
$\vdots$	$\vdots$	$\vdots$
$16,\ldots,13$	$16,\ldots,19$	15
Resulting samples: $\text{Sample}_0,\ldots,\text{Sample}_3,\text{Sample}_{31},\ldots,\text{Sample}_{29}$		

$V^{even}_{64,\ldots,127}$ DCT32 output index	$V^{odd}_{64,\ldots,127}$ DCT32 output index	DCT32 output age
$20,\ldots,23$	$12,\ldots,9$	0
$12,\ldots,9$	$20,\ldots,23$	1
$20,\ldots,23$	$12,\ldots,9$	2
$\vdots$	$\vdots$	$\vdots$
$12,\ldots,9$	$20,\ldots,23$	15
Resulting samples: $\text{Sample}_4,\ldots,\text{Sample}_7,\text{Sample}_{28},\ldots,\text{Sample}_{25}$		

$V^{even}_{128,\ldots,191}$ DCT32 output index	$V^{odd}_{128,\ldots,191}$ DCT32 output index	DCT32 output age
$24,\ldots,27$	$8,\ldots,5$	0
$8,\ldots,5$	$24,\ldots,27$	1
$24,\ldots,27$	$8,\ldots,5$	2
$\vdots$	$\vdots$	$\vdots$
$8,\ldots,5$	$24,\ldots,27$	15
Resulting samples: $\text{Sample}_8,\ldots,\text{Sample}_{11},\text{Sample}_{24},\ldots,\text{Sample}_{21}$		

$V^{even}_{192,\ldots,271}$ DCT32 output index	$V^{odd}_{192,\ldots,271}$ DCT32 output index	DCT32 output age
$28,\ldots,31,zero$	$4,\ldots,0$	0
$4,\ldots,0$	$28,\ldots,31,zero$	1
$28,\ldots,31,zero$	$4,\ldots,0$	2
$\vdots$	$\vdots$	$\vdots$
$4,\ldots,0$	$28,\ldots,31,zero$	15
Resulting samples: $\text{Sample}_{12},\ldots,\text{Sample}_{16},\text{Sample}_{20},\ldots,\text{Sample}_{17}$		

Table 6.3: The final layout of V. Note that *zero* is the value 0, not the index 0.

6.2 Decoder verification

There are several test bitstreams available to verify that a decoder is compliant [4] to the MP3 standard. The accuracy of the decoder was verified by decoding *compl.bit*. The output of the decoder was found to comply with the requirements of a limited accuracy MPEG-1 layer III decoder. The RMS error compared to the reference output was $3.2 \cdot 10^{-5}$. This is similar to the accuracy predicted by the initial experiments shown in figure 3.1.

The other test bitstreams were used for verifying that the decoder could handle all kinds of bitstream parameters. For example, there are test bitstreams that contain all kinds of header bits, all kind of bit rates, the different stereo modes, different flags and all Huffman codes.

The assembly language implementation was debugged by dumping all memory accesses to a file. These values were compared with the corresponding value in the C implementation. This was relatively straight forward as the assembly language implementation is very similar to the C implementation.

Core algorithms like the IMDCT and the subband synthesis were debugged by linking the simulator with dedicated test code. This test code compared the values of the assembly language implementation to values calculated by C models using double precision floating point arithmetics.

6.3 Listening test

A listening test was performed where various samples were encoded with LAME. The source material was gathered from the *Sound quality assessment material* — SQAM [10], various test files from the LAME project [11] and finally some samples were gathered from CDs.

The SQAM samples include sounds from several different instruments and speech in German, English and French.

The LAME test files are samples that have caused problems for LAME. These represent a wide variety of music genres and other sounds.

The samples from CDs were gathered from a Roxette album, a Secret Garden album and finally from an album with Brahms' Piano Concerto No. 2.

Only one of the 200 files examined could be distinguished from a version decoded by a floating point decoder. This file was encoded at 32 kbit/s and it did not sound very good in the first place.

7

Benchmarks and profiling

This chapter contains information about the performance of the implemented decoder.

7.1 Clock frequency requirements

Custom test bitstreams were created to benchmark the decoder. The worst case performance is obtained with 48 kHz sample rate, a bit rate of 320 kbit/s, joint-stereo, only short blocks and values optimized to make the sample dequantization difficult by avoiding values that are handled by fast special cases. The decoder is able to decode such a bitstream in real-time if the clock frequency is 20 MHz. A few other combinations are listed in table 7.1.

Test bitstream	**MIPS (Worst case)**	**MIPS (Average)**
48 kHz, 320 kbit/s, joint-stereo	20	18
44.1 kHz, 128 kbit/s, joint-stereo	15	14
44.1 kHz, 64 kbit/s, mono	7	7

Table 7.1: The worst case and average case performance of the decoder. The MIPS value assumes that the external sample FIFO is large enough to mitigate the impact of a large bit reservoir.

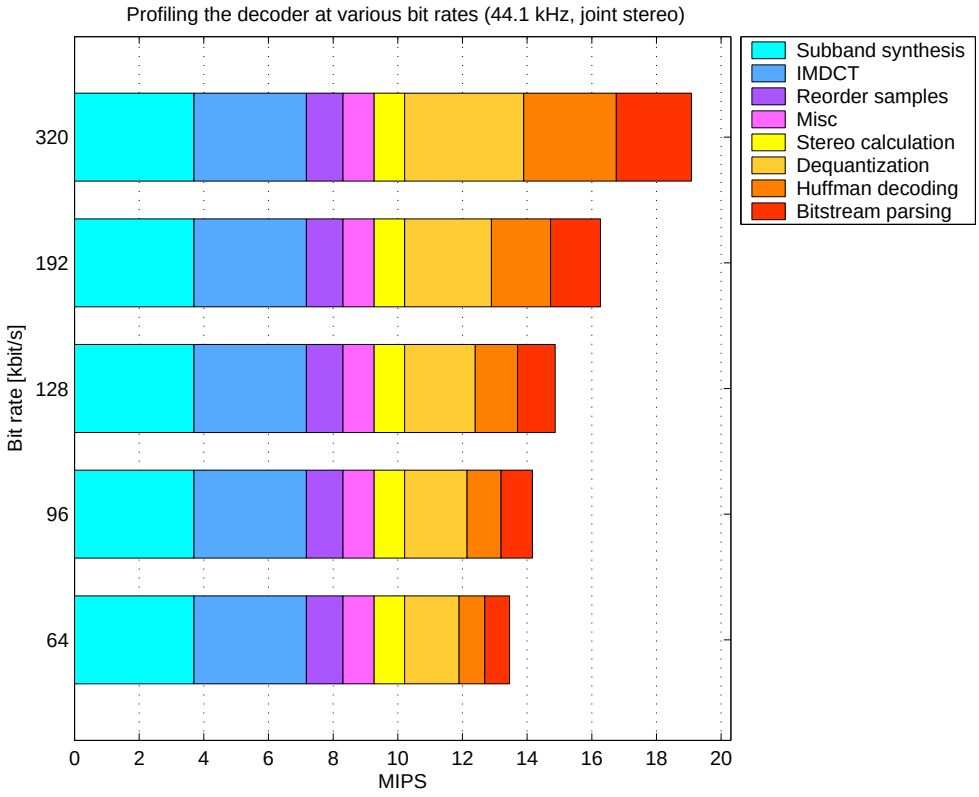


Figure 7.1: Performance of the decoder while decoding joint-stereo streams with different bit rates.

It is not necessary to take the bit reservoir into account when calculating the absolute worst case performance since the MP3 standard does not allow the usage of a bit reservoir at 320 kbit/s.

The performance of the different components of the decoder while decoding a joint-stereo file is listed in figure 7.1. The MIPS usage is reduced by about 50% if a mono file is decoded as seen in figure 7.2. This is expected since the decoder only has to run the IMDCT and subband synthesis stage for one channel. These figures are created for a worst case bitstream at the specific bit rate and sampling frequency.

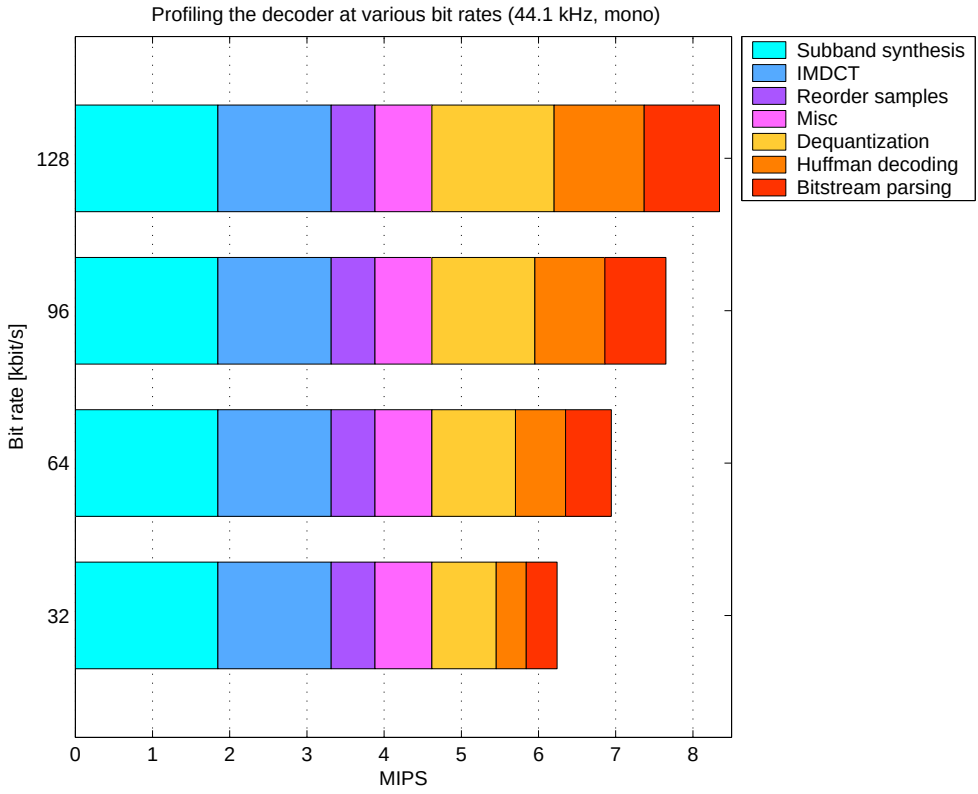


Figure 7.2: Performance of the decoder while decoding mono streams with different bit rates.

The figures also illustrates that the time consumption of some parts of the decoder is static provided that only the bit rate varies. The only parts that are dynamic in this case are the Huffman decoding, the bit stream parsing and the dequantization.

7.2 Memory usage

The total data memory usage of the decoder is shown in table 7.2 and the total program memory usage is shown in table 7.3. The main reason for the large program memory is that most performance critical loops are unrolled. The Huffman decoder is also responsible for a significant part of the program memory.

The constant memory is mainly used for coefficients for various transforms and windowing operations. Table 7.4 shows the usage of the constant memory. In addition, an external sample FIFO has to be added. The size of that FIFO is determined by how fast the CPU can fill it, that is, it depends on the clock frequency. It should contain at least 576 stereo samples if the decoder is clocked at 20 MHz. A higher clock speed will reduce the necessary storage space.

Part	**Words (16-bit)**
Bitstream buffer	1024
IMDCT overlap buffers	3456
Subband synthesis	1088
PCM buffer	64
Temporary variables, bitstream parameters, etc	397
Total	6029

Table 7.2: Allocation of data memory.

Part	**Words (24-bit)**
Huffman decoding	2954
Subband synthesis	1029
Bitstream parsing	908
IMDCT	709
Misc	454
Stereo calculation	392
Dequantization	342
Total	6788

Table 7.3: Allocation of program memory.

Part	**Words (23-bit)**
Sample dequantization	74
IMDCT filter bank	212
Subband Synthesis	576
Misc	46
Total	908

Table 7.4: Allocation of constant memory.

7.3 Instruction usage statistics

The instruction usage is shown in table 7.5. The profiling data was gathered while decoding *compl.bit*. The immediate form of add is the most common instruction because the add instructions are used for updating pointers and loop counters. A loop instruction and a ld and st instruction with post increment could therefore improve performance by about 10%.

The fmac and fmaci instructions use the floating point adder, multiplier and the memory. As fmac does not increment the address pointer, the content of that address could be buffered as long as fmac follows fmac or fmaci. This is almost always the case. A fair approximation is that while fmaci accesses memory, fmac does not.

The memory utilization is about 23.4%, the FPU usage is 32.1%.

FPU operations	**Cycles**
fadd, fsub	9.7%
fmul, fmulc	5.1%
fint, fpack	4.1%
Total	19%

ALU operations	**Cycles**
addi	13.6%
add, sub, subi	2.7%
and, or, xor, andi, ori, xori	2.2%
Total	19%

MAC operations	**Cycles**
fmac	6.2%
fmaci	7.0%
Total	13%

Load & store	**Cycles**
ld, ldi, ldf	8.5%
st, sti	7.9%
Total	16%

Branches	**Cycles**
bnez, beqz	11.1%
calli, call, ret	6.7%
bra, jmp	
bnbc, bnbs	1.5%
Total	19%

Misc	**Cycles**
set, seth	3.6%
rdsr, wrsr, setsr	2.6%
ltoh, htol	0.8%
nbit	0.4%
nop (Pipeline delay)	6.6%
Total	14%

Table 7.5: Cycle usage of various instructions.

8

RTL implementation

This chapter describes the RTL implementation and the FPGA prototype.

8.1 VHDL

A *register transfer level*, RTL, model of the CPU was written in VHDL. The architecture of the processor was easy to implement in VHDL. No real effort was made to optimize the design.

8.1.1 Development environment

The development environment was FPGA Advantage from Mentor Graphics. Emacs was used as a VHDL editor. Xilinx' place & route was used for synthesizing the design.

8.1.2 Functional verification

To verify that the VHDL implementation was correct a variety of test cases were written. A separate HDL level test case verified the FPU functionality. Only corner cases and a selection of random input values were tested as it would take a huge amount of time to test all possible input combinations.

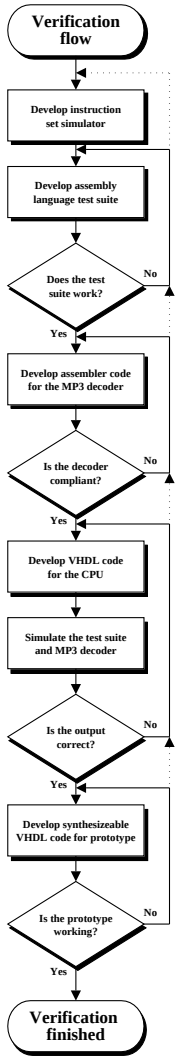


Figure 8.1: The system verification flow.

An instruction level test suite was written as well. All instructions are tested, but not all operand values. New test cases were added as bugs were found and corrected in the VHDL code.

Finally, the MP3 decoder was run on the compliance test bitstream and the output was compared to the output of the instruction set simulator.

The overall system verification flow is shown in figure 8.1.

8.2 FPGA prototype

The VHDL implementation was tested on a prototype board called XSV-300. The board is developed by Xess and is based on a Virtex-300 FPGA. (Xess has discontinued the production of this board.) The board has two SRAM banks with 16-bit data busses. Each bank store 8 megabits. An on-board stereo audio codec is available as well. In addition, the board features a large amount of other peripheral devices. The block schematics of the board is shown in figure 8.2.

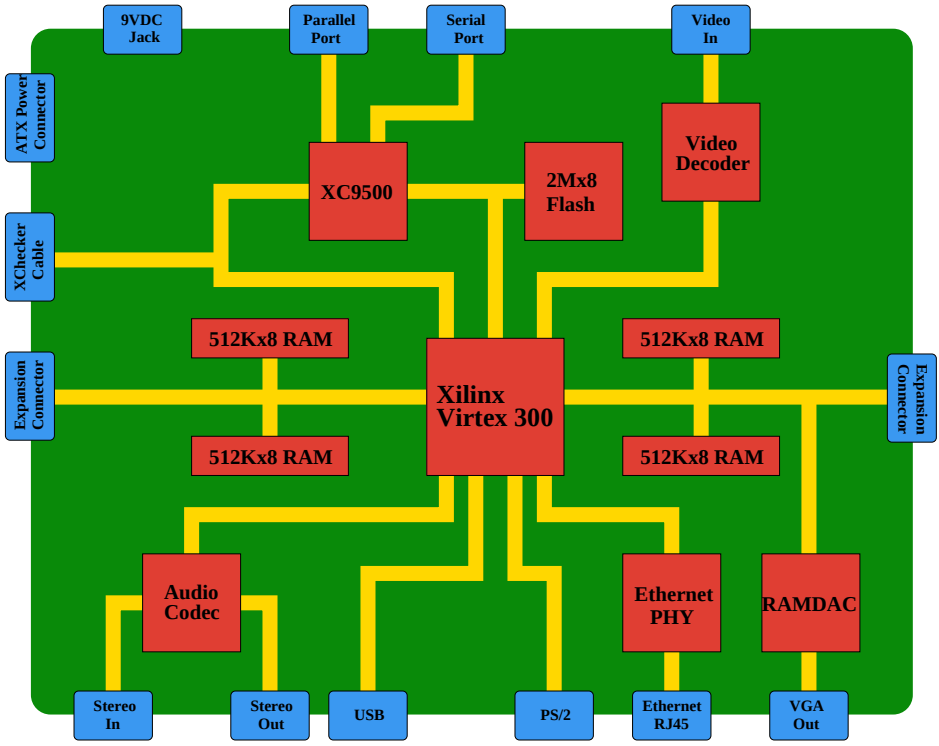


Figure 8.2: Block diagram of the XSV-300.

8.2.1 Resource usage

The Virtex-300 FPGA has 64 Kbit RAM available internally, divided into 16 dual port blocks. Since this is not enough for the MP3 decoder, external SRAM was also used. One external SRAM bank was dedicated to the program memory. Data was read on both clock edges in order to

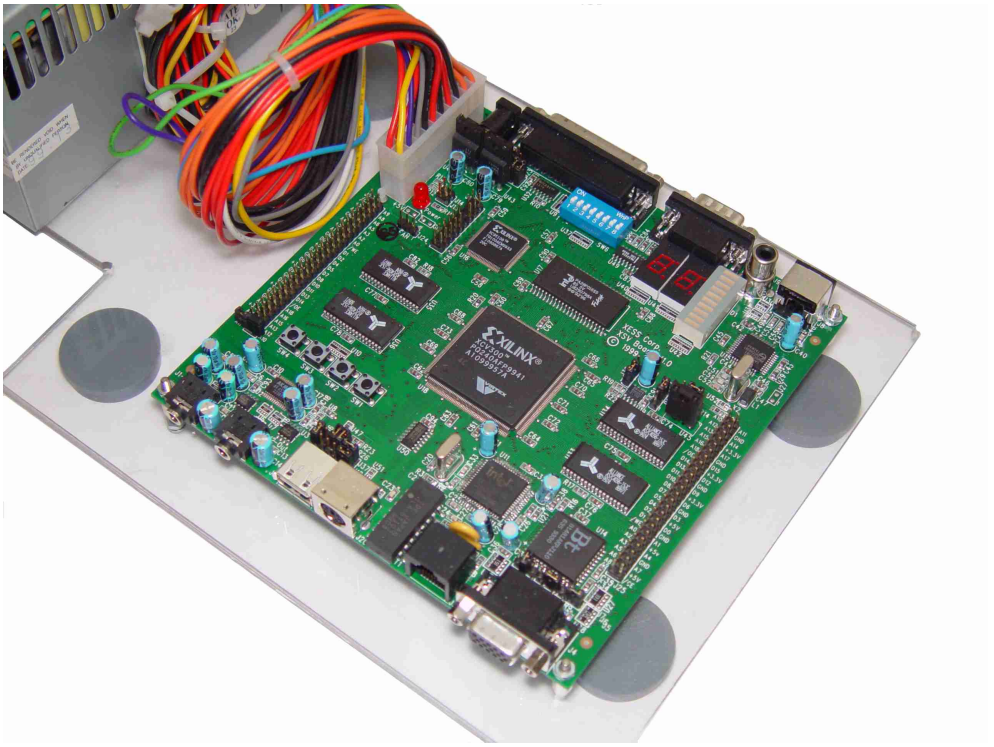


Figure 8.3: The XSV-300 prototype card.

get a 24-bit instruction on every cycle. (The remaining 8 bits were not used.) The other SRAM bank was used as the data memory of the decoder. The MP3 bitstream was stored in this RAM as well. The constant memory used 6 block RAMs of the Virtex-300. Finally, a sample FIFO was implemented using 8 block RAMs.

8.2.2 FPGA resource usage

The FPGA usage was about 30%. The prototype was clocked at 20 MHz. It could run faster according to the synthesizer but the design does not work at 25 MHz, most likely because of the interface to the external memory banks.

The resource usage of various parts of the design is shown in figure 8.4. The critical path was located in the logarithmic shifter of the FPU. This was no surprise as FPGAs are not very suited for the large multiplexers

needed by a fast shifter. The heavy pipelining of the design ensured that the design could easily be synthesized without any FPGA specific optimizations in the VHDL code.

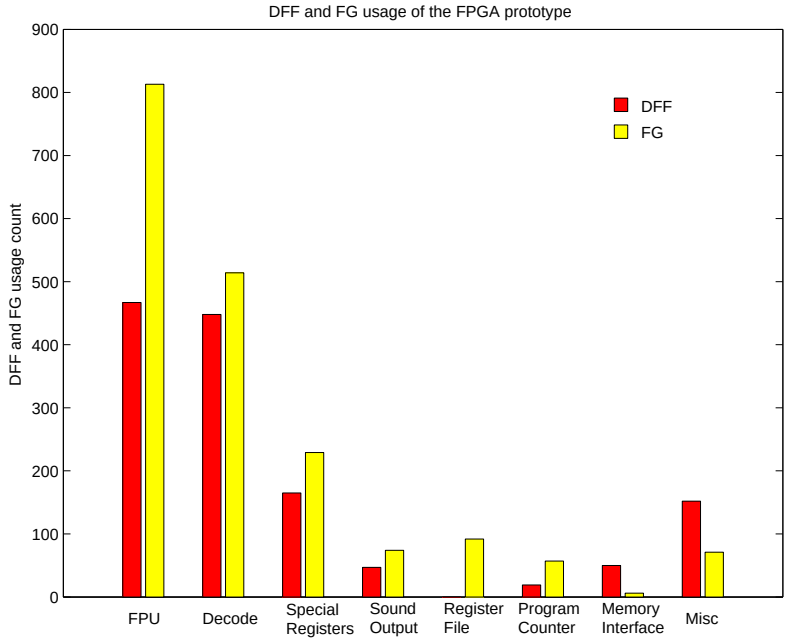


Figure 8.4: The utilization of the FPGA. DFF is a regular flip flop and an FG is a 4-bit function generator. FGs can also be used for implementing register files in an area efficient manner.

9

Results

Overall, the project has been a success. The decoder has successfully decoded all tested streams and the performance is satisfactory, especially considering the limited instruction set. The CPU was easy to implement in VHDL but somewhat awkward to program for, especially in the beginning.

- The decoder stores intermediate data in a 16-bit floating point format to limit memory usage.
- The decoder is verified as a limited accuracy ISO/IEC 11172-3 MPEG-1 layer III decoder. It does not support layer I or II. The RMS of error is $3.2 \cdot 10^{-5}$.
- A clock frequency of 20 MHz is enough to decode all MPEG-1 layer III streams.
- VHDL code for the hardware has been implemented and verified on an FPGA prototype board.
- The gate count, excluding external memories, is 32500 gates when synthesized against Leonardo Spectrum's sample *SCL05u* technology.
- The size of the program memory is 6785 24-bit words. The size of the constant memory is 908 23-bit words. The size of the data memory is 6069 16-bit words.

10

Future work

This chapter describes various improvements that can be made to the software and the hardware.

10.1 Improved software

It would be relatively straight-forward to add layer I and II to the current decoder. It would also be interesting to investigate if other audio formats such as AAC and Ogg Vorbis can be decoded using low precision floating point arithmetics.

10.2 Improved hardware

It is unlikely that large gains could be made without modifying the hardware architecture since the software is already optimized.

- Data memory access with *pointer auto-increment* would improve performance and reduce the size of the program memory.
- A hardware loop instruction would reduce register pressure and the size of the program memory. In the current implementation the critical loops are unrolled.
- There is a large overhead associated with the Huffman decoder. The tree can be represented with 10 bits per node whereas the current implementation uses one instruction per node (24 bits).

- Optional result forwarding would eliminate some explicit pipeline stalls.
- The FPU should have a *saturation on overflow* mode. (It is possible to construct bitstreams that will cause an overflow in fpack.)

There are a few other optimizations that could be useful if the CPU is going to be used for other applications.

- A bit-reversed addressing mode for FFT and other transforms.
- The conditional branches should be increased beyond *branch-if-zero* and *branch-if-not-zero*.
- Support for hardware interrupts.

10.3 Improved development tools

The assembler supports only a limited form of expressions, in the form label+offset. This could be extended so that arbitrary expressions could be used.

It is sometimes necessary to declare a function before it is used due to improper handling of forward references across name space levels.

The speed of the instruction set simulator could be improved. An interactive debugger could also be implemented to aid development.

10.4 Power measurements

It would be interesting to measure the power consumption of the hardware. It would be interesting to compare this to other MP3 decoding solutions.

A

Instruction set reference

This chapter contains the instruction set reference.

Notation

x << y	Shift x left by y steps.
abs16	16-bit absolute address contained in the instruction word.
cm(x)	The floating point value located in constant memory address x.
fp16to23(x)	Expand x from a 16-bit floating point value to a 23-bit floating point value.
fp23toint(x)	Convert the 23-bit floating point value x, scaled by 32768, to a signed integer, with saturation.
fp(x)	x is interpreted as a 23-bit floating point argument.
fp23to16(x)	Convert x from a 23-bit floating point value to a 16-bit floating point value with rounding.
imm7	7-bit immediate signed data contained in the instruction word.
imm10	10-bit immediate signed data contained in the instruction word.
imm16	16-bit immediate signed data contained in the instruction word.
highbits(x)	Refers to the upper 7 bits of the register x.
msb(x)	Refers to the most significant bit of x.

pop(x) Pop an element off the hardware stack.

push(x) Push an element onto the hardware stack.

rD An arbitrary register used as a destination argument. This refers to the lower 16 bits of the register unless *fp*() or *highbits*() is used.

r0...r15 Register 0...15.

rA,rB An arbitrary register used as a source argument. This refers to the lower 16 bits of the register unless *fp*() or *highbits*() is used.

signextend(x) Sign extend x to the appropriate number of bits.

sr0...sr15 Special register 0...15.

datamem(x) Value of data memory location x.

An operation surrounded by parentheses in the pipeline diagram is not always executed, e.g., "(Update PC)" in a conditional branch.

abort

Fetch

Decode

Assembler syntax:

abort

Operation:

Abort simulation.

The instruction set simulator will abort simulation when this instruction is encountered. The behaviour of this instruction is undefined on the real hardware.

add

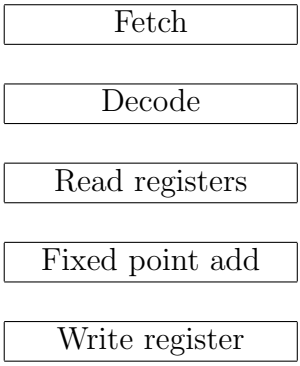
Assembler syntax:

add rA,rB,rD

Operation:

rA + rB → rD

16-bit fixed point addition.



addi

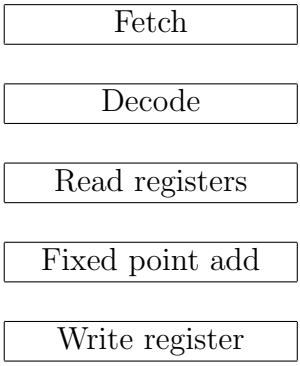
Assembler syntax:

addi #imm10,rB,rD

Operation:

signextend(imm10) + rB → rD

16-bit fixed point addition, immediate data.



and

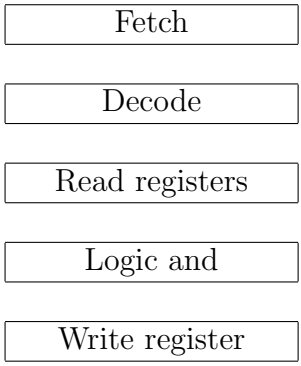
Assembler syntax:

and rA,rB,rD

Operation:

rA $\wedge$ rB $\rightarrow$ rD

16-bit bitwise and.



andi

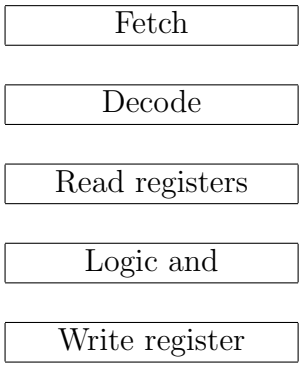
Assembler syntax:

andi #imm10,rB,rD

Operation:

signextend(imm10) $\wedge$ rB $\rightarrow$ rD

16-bit bitwise and, immediate data.



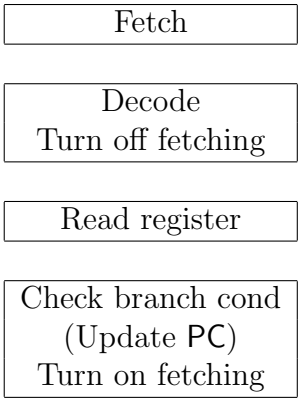
beqz

Assembler syntax:

beqz rA,abs16

Operation:

if rA = 0
    abs16 → PC



Conditional jump to address specified by abs16.
Jump is taken if rA is zero. There is one delay slot, i.e., one instruction after the jump is executed. (See bra for a delay slot example.)

The results of having a branch instruction (including ret) in the delay slot is undefined.

bnbc

Fetch

Assembler syntax:

bnbc abs16

Decode
Turn off fetching
nop→next insn

Operation:

Update sr0
Update sr1
(Update sr2)

(Memory read)
Check branch cond
(Update PC)
Turn on fetching

nop → next instruction
msb(sr0) → next_bit
(sr0 << 1) → sr0
if sr1 = 0
    datamem(sr2) → sr0
    15 → sr1
    if sr2 = sr4
        sr3 → sr2
    else
        sr2 + 1 → sr2
else
    sr1 − 1 → sr1
if next_bit = 0
    abs16 → PC

NOTE: *The instruction following the bnbc instruction is ignored.*

Read the next bit from the bit buffer and branch to abs16 if it is clear.

The instruction that follows bnbc is changed to a nop. This makes it possible to construct small Huffman tables.

The bitstream is located in RAM, but the CPU buffers 16 bits (one memory word) to reduce the number of memory accesses. When the 16 bits in this bit cache have been read, a new word is read from memory. All of this is done automatically by the hardware.

sr0 contains the bit buffer and sr1 is set to the number of bits remaining, minus one. The most significant bit in sr0 is always read and the contents of sr0 is shifted left one bit on each read while sr1 is decremented. When sr1 indicates that there are no bits left in sr0, a new word is read from RAM at the address indicated by sr2. sr2 is then incremented unless it equals sr4, in which case it is set to the contents of sr3.

bnbs

Fetch

Assembler syntax:

bnbs abs16

Decode
Turn off fetching
nop→next insn

Operation:

nop → next instruction
msb(sr0) → next_bit
(sr0 << 1) → sr0
if sr1 = 0
    datamem(sr2) → sr0
    15 → sr1
    if sr2 = sr4
        sr3 → sr2
    else
        sr2 + 1 → sr2
else
    sr1 − 1 → sr1
if next_bit = 1
    abs16 → PC

Update sr0
Update sr1
(Update sr2)

(Memory read)
Check branch cond
(Update PC)
Turn on fetching

NOTE: *The instruction following the bnbc instruction is ignored.*

Read the next bit from the bit buffer and branch to abs16 if it is set.

The instruction that follows bnbs is changed to a nop. This makes it possible to construct small Huffman tables.

The bitstream is located in RAM, but the CPU buffers 16 bits (one memory word) to reduce the number of memory accesses. When the 16 bits in this bit cache have been read, a new word is read from memory. All of this is done automatically by the hardware.

sr0 contains the bit buffer and sr1 is set to the number of bits remaining, minus one. The most significant bit in sr0 is always read and the contents of sr0 is shifted left one bit on each read while sr1 is decremented. When sr1 indicates that there are no bits left in sr0, a new word is read from RAM at the address indicated by sr2. sr2 is then incremented unless it equals sr4, in which case it is set to the contents of sr3.

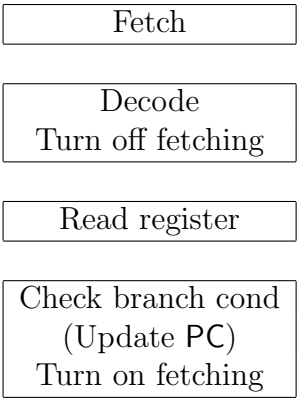
bnez

Assembler syntax:

bnez rA,abs16

Operation:

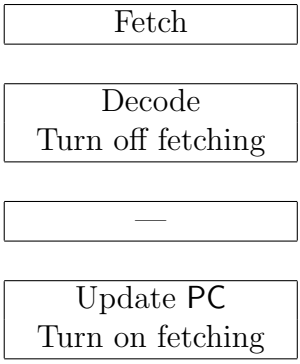
if rA $\neq$ 0
abs16 $\rightarrow$ PC



Conditional jump to address specified by abs16.
Jump is taken if rA is non zero. There is one delay slot, i.e., one instruction after the jump is executed. (See bra for a delay slot example.)

The results of having a branch instruction (including ret) in the delay slot is undefined.

bra



Assembler syntax:

bra abs16

Operation:

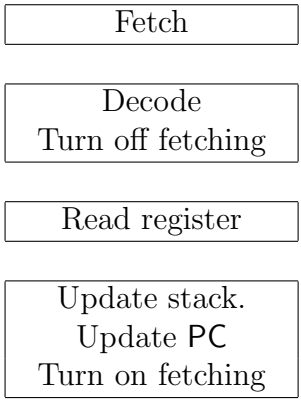
abs16 → PC

Unconditional jump to address abs16. There is one delay slot, i.e., one instruction after the jump is executed. The results of having a branch instruction (including ret) in the delay slot is undefined.

Delay slot example:

```
        bra   foobar
        set   #0x0,r0   ; This will be executed (delay slot)
        set   #0x1,r1   ; This will not be executed
foobar:
        set   #0x2,r2   ; Execution continues here
```

call



Assembler syntax:

call rA

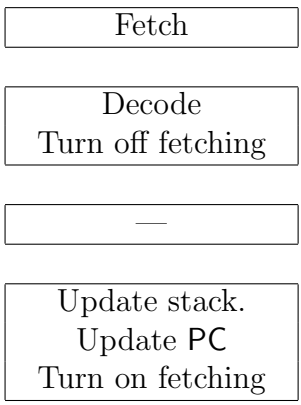
Operation:

push(next PC)
rA → PC

Unconditional call to subroutine specified by the low 16 bits of rA. There is one delay slot, i.e., one instruction after the jump is executed. (See bra for a delay slot example.)

The results of having a branch instruction (including ret) in the delay slot is undefined.

calli



Assembler syntax:

calli abs16

Operation:

push(next PC)
abs16 → PC

Unconditional call to subroutine specified by abs16. There is one delay slot, i.e., one instruction after the jump is executed. (See bra for a delay slot example.)

The results of having a branch instruction (including ret) in the delay slot is undefined.

fadd

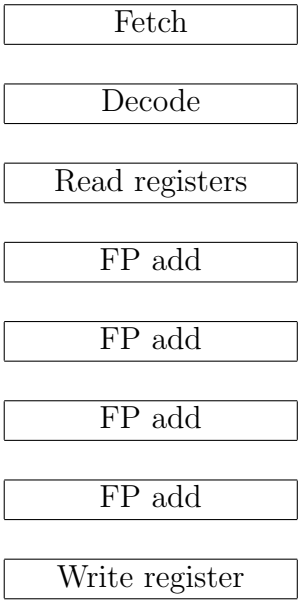
Assembler syntax:

fadd rA,rB,rD

Operation:

fp(rA) + *fp*(rB) → *fp*(rD)

Floating point addition.



fint

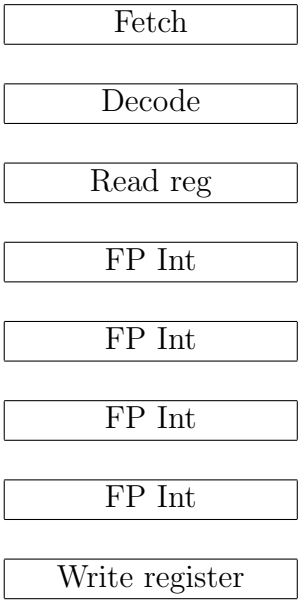
Assembler syntax:

fint rA,rD

Operation:

fp23toint(rA) → *fp*(rD)

Scale a 23-bit fp by 32768 and convert it to an integer.



fmac

Assembler syntax:

fmac rA,rD

Operation:

fp(fmulbuf) + *fp*(rA) → *fp*(rD)
fp(fmembuf) · *fp*(*cm*(sr5))→ *fp*(fmulbuf)
fp16to23(*datamem*(sr2)) → *fp*(fmacbuf)
sr5 + 1 → sr5

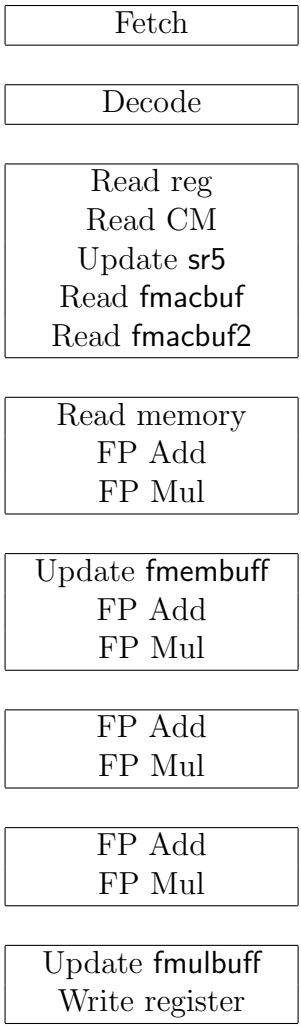
NOTE: *These are not executed in sync!*
See pipeline figure.

Floating point multiply and accumulate.

The fmac instruction combines 3 operations (load, multiply, accumulate) into one instruction.

A *group* of fmac instructions will perform the following operations, read data memory at address sr2, expand it to a high precision floating point number, multiply it with a constant fetched from constant memory location sr5, accumulate that result with register rA into register rD. Increment sr5.

See fmaci for an example.



fmaci

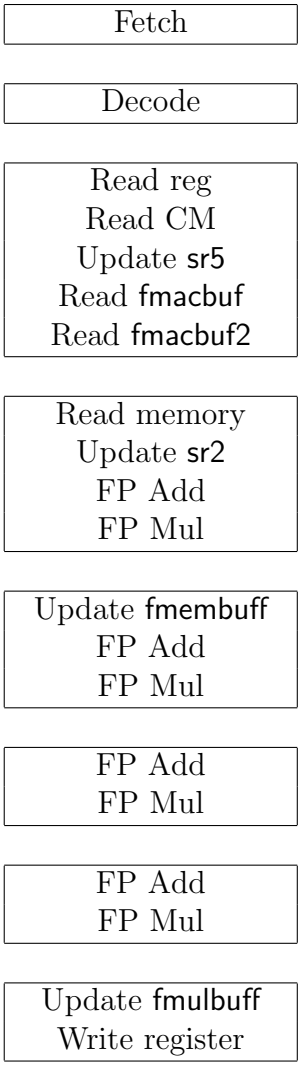
Assembler syntax:

fmaci rA,rD

Operation:

fp(fmulbuf) + *fp*(rA) → *fp*(rD)
fp(fmembuf) · *fp*(*cm*(sr5)) → *fp*(fmulbuf)
fp16to23(*datamem*(sr2)) → *fp*(fmacbuf)
sr5 + 1 → sr5
if sr2 = sr4
    sr3 → sr2
else
    sr2 + 1 → sr2

NOTE: *These are not executed in sync!*
See pipeline figure.



Floating point multiply and accumulate.

The fmaci instruction combines 4 operations (load, multiply, accumulate, modulo increment) into one instruction.

A *group* of fmaci instructions will perform the following operations, read data memory at address sr2, expand it to a high precision floating point number, multiply it with a constant fetched from constant memory location sr5, accumulate that result with register rA into register rD. Increment sr5. Increment sr2 with 1 unless sr2 is equal to sr4. Set sr2 to sr3 in that case.

An example of how to use the fmac and fmaci instruction is included on the next page.

fmac, fmaci example:

This is an example of the following operation:

r0= $\sum_{k=0}^{2}$ *fp*(*cm*(cmconsts+0+6·k)) · *fp16to23*(*datamem*(startptr+0))

r1= $\sum_{k=0}^{2}$ *fp*(*cm*(cmconsts+1+6·k)) · *fp16to23*(*datamem*(startptr+1))

r2= $\sum_{k=0}^{2}$ *fp*(*cm*(cmconsts+2+6·k)) · *fp16to23*(*datamem*(startptr+2))

r3= $\sum_{k=0}^{2}$ *fp*(*cm*(cmconsts+3+6·k)) · *fp16to23*(*datamem*(startptr+3))

r4= $\sum_{k=0}^{2}$ *fp*(*cm*(cmconsts+4+6·k)) · *fp16to23*(*datamem*(startptr+4))

r5= $\sum_{k=0}^{2}$ *fp*(*cm*(cmconsts+5+6·k)) · *fp16to23*(*datamem*(startptr+5))

```
; A program demonstrating how to calculate
; the above values using fmac/fmaci
setsr  #cmconsts−2,sr5  ; Constant memory pointer

; Data memory pointers:
setsr  #startptr,sr2    ; Starting location
setsr  #startptr,sr3    ; Wraparound address
setsr  #endptr,sr4      ; End address

fsub   r5,r5            ; zero out accumulators

; fmembuf is not valid
; fmulbuf is not valid
; constant memory pointer sr5 is set to cmconsts−2
fmac   r0,r0            ; Write scratch value to r0, start
fmac   r0,r0            ; loading values from data memory

; fmembuf is now valid
; Constant memory pointer sr5 is now at cmconsts
fmac   r0,r0            ; Start multiplying values valid values
fmac   r0,r0            ; (Still writing undefined values to r0)
fmac   r0,r0
fmaci  r0,r0            ; increment data memory read pointer

; fmembuf and fmulbuf are now both valid
; Constant memory pointer sr5 is now at comconsts+4
```

```
fmac  r5,r0          ; start accumulating, r5 contains 0 at this point
fmac  r5,r1
fmac  r5,r2
fmac  r5,r3
fmac  r5,r4
fmaci r5,r5          ; increment data memory read pointer

fmac  r0,r0          ; r0 is now datamem(startptr)*cm(cmconstptr)
fmac  r1,r1
fmac  r2,r2
fmac  r3,r3
fmac  r4,r4
fmac  r5,r5          ; No need to increase data memory pointer now

fmac  r0,r0          ; r0 is now datamem(startptr)*cm(cmconstptr) +
fmac  r1,r1          ;      datamem(startptr+1)*cm(cmconstptr+6)
fmac  r2,r2          ; No more valid values are fed to multiplier now
fmac  r3,r3
fmac  r4,r4
fmac  r5,r5

fpack r0,r0          ; r0 is now datamem(startptr)*cm(cmconstptr) +
fpack r1,r1          ;      datamem(startptr+1)*cm(cmconstptr+6) +
fpack r2,r2          ;      datamem(startptr+2)*cm(cmconstptr+12)
```

fmul

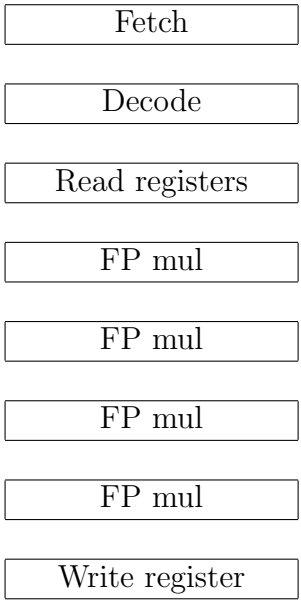
Assembler syntax:

fmul rA,rB,rD

Operation:

$fp(\text{rA}) \cdot fp(\text{rB}) \rightarrow fp(\text{rD})$

Floating point multiplication.



fmulc

Assembler syntax:

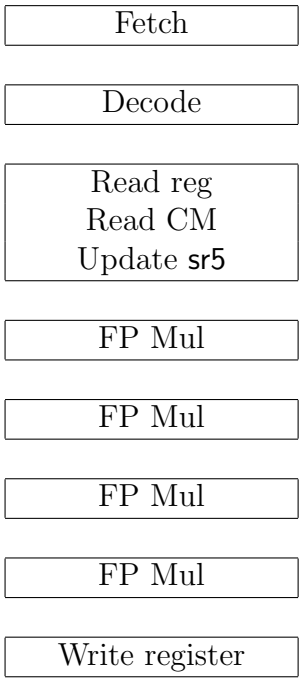
fmulc rA,rD

Operation:

$fp(\text{rA}) \cdot fp(cm(\text{sr5})) \rightarrow fp(\text{rD})$

$\text{sr5} + 1 \rightarrow \text{sr5}$

Floating point multiplication with constant. The constant memory is addressed with sr5. sr5 is auto-incremented.



fpack

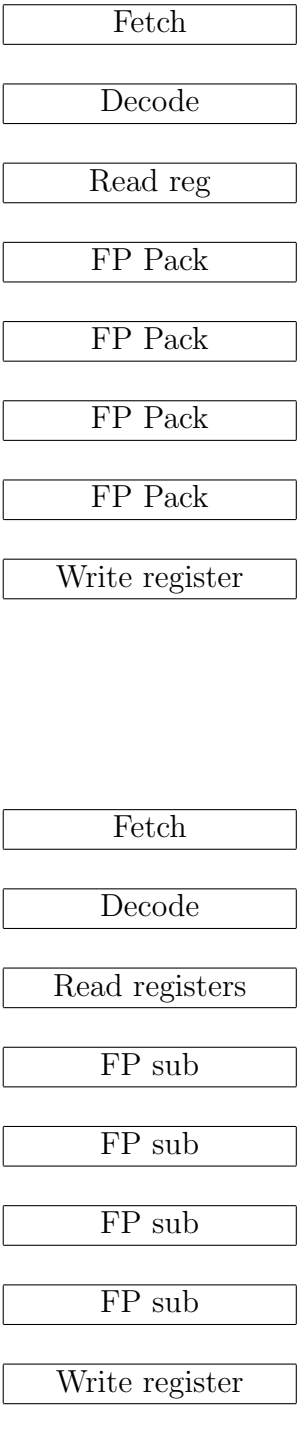
Assembler syntax:

fpack rA,rD

Operation:

fp23to16(*fp*(rA)) → rD

Convert a high precision 23-bit fp to a low precision 16-bit fp value with rounding. A value with an exponent that is smaller than the 16-bit fp format allows is rounded to zero. The behaviour is undefined if a value that is too large is passed to fpack. (The instruction set simulator will abort and notify the user of an error condition in this case.)



fsub

Assembler syntax:

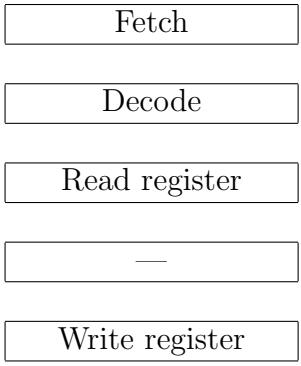
fsub rA,rB,rD

Operation:

fp(rA) − *fp*(rB) → *fp*(rD)

Floating point subtraction.

htol



Assembler syntax:

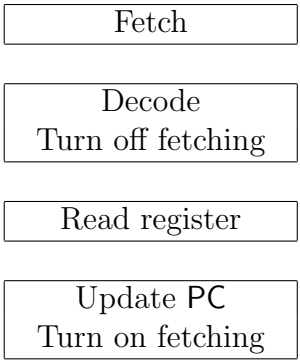
htol rA,rD

Operation:

highbits(rA) → rD

Transfer high bits of rA to low bits of rD. The 9 most significant bits of rD are set to 0.

jmp



Assembler syntax:

jmp rA

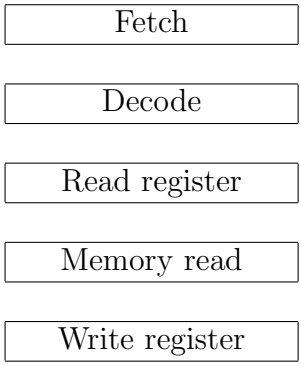
Operation:

rA → PC

Unconditional jump to address specified by the low 16 bits of rA. There is one delay slot, i.e., one instruction after the jump is executed. (See bra for a delay slot example.)

The results of having a branch instruction (including ret) in the delay slot is undefined.

ld



Assembler syntax:

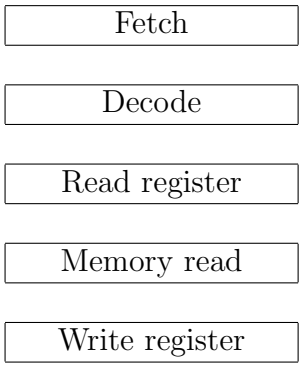
ld [rA],rD

Operation:

datamem(rA) → rD

Load low 16 bits from data memory, register indirect addressing.

ldf



Assembler syntax:

ldf [rA],rD

Operation:

fp16to23(*datamem*(rA)) → rD

Load a floating point number from memory, register indirect addressing, and expand it from a 16-bit low precision fp to 23-bit high precision fp.

ldi

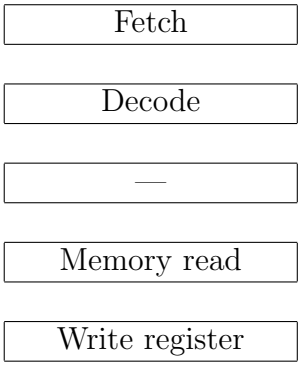
Assembler syntax:

ldi [abs16],rD

Operation:

datamem(abs16) → rD

Load low 16 bits from data memory, absolute addressing.



ltoh

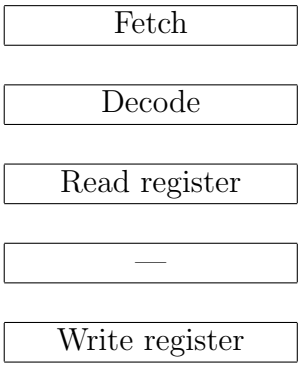
Assembler syntax:

ltoh rA,rD

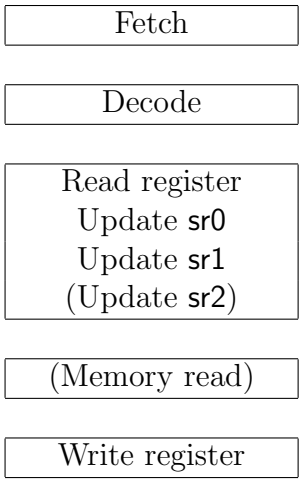
Operation:

rA → *highbits*(rD)

Transfer low bits of rA to high bits of rD.



nbit



Assembler syntax:

nbit rA,rD

Operation:

msb(sr0) → next_bit
(sr0 << 1) → sr0
(rA << 1) ∨ next_bit → rD
if sr1 = 0
    datamem(sr2) → sr0
    15 → sr1
    if sr2 = sr4
        sr3 → sr2
    else
        sr2 + 1 → sr2
else
    sr1 − 1 → sr1

Read the next bit from the bit buffer.

The bitstream is located in RAM, but the CPU buffers 16 bits (one memory word) to reduce the number of memory accesses. When the 16 bits in this bit cache have been read, a new word is read from memory. All of this is done automatically by the hardware.

sr0 contains the bit buffer and sr1 is set to the number of bits remaining, minus one. The most significant bit in sr0 is always read and the contents of sr0 is automatically shifted left one bit on each read while sr1 is decremented. When sr1 indicates that there are no bits left in sr0, a new word is read from RAM at the address indicated by sr2. sr2 is then incremented unless it equals sr4, in which case it is set to the contents of sr3.

nop

Fetch

Decode

Assembler syntax:

nop

Operation:

None

No operation.

or

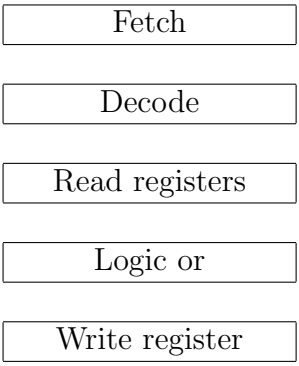
Assembler syntax:

or rA,rB,rD

Operation:

rA $\vee$ rB $\rightarrow$ rD

16-bit bitwise inclusive or.



ori

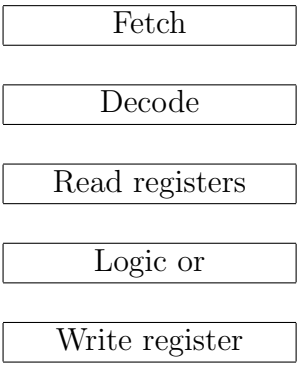
Assembler syntax:

ori #imm10,rB,rD

Operation:

signextend(imm10) $\vee$ rB $\rightarrow$ rD

16-bit bitwise inclusive or, immediate data.



rdsr

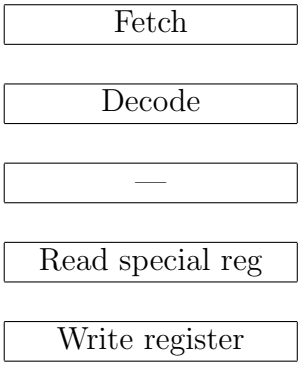
Assembler syntax:

rdsr srA,rD

Operation:

srA → rD

Write the value of special register srA to destination register rD.



ret

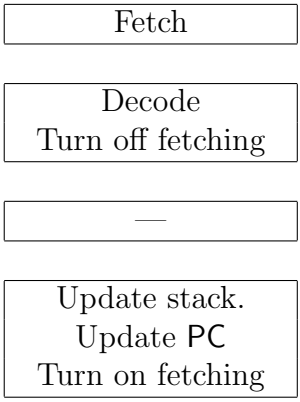
Assembler syntax:

ret

Operation:

pop(PC)

Return from subroutine. There is one delay slot, i.e., one instruction after the jump is executed. The results of having a branch instruction (including ret) in the delay slot is undefined. (See bra for a delay slot example.)



set

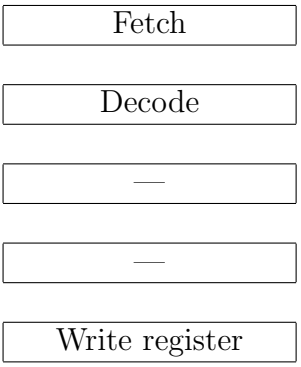
Assembler syntax:

set #imm16,rD

Operation:

imm16 → rD

Set low 16 bits from immediate data.



seth

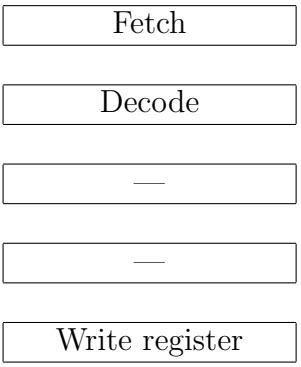
Assembler syntax:

seth #imm7,rD

Operation:

imm7 → *highbits*(rD)

Set high 7 bits from immediate data.



setsr

Assembler syntax:

setsr #imm16,srD

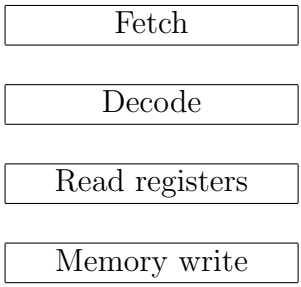
Operation:

imm16 → srD

Set special register srD from immediate data.

Fetch
Decode
—
Write special reg

st



Assembler syntax:

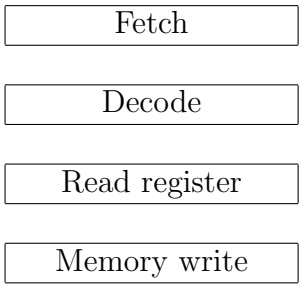
st [rD],rA

Operation:

rA → *datamem*(rD)

Store low 16 bits to data memory, register indirect addressing.

sti



Assembler syntax:

sti [abs16],rA

Operation:

rA → *datamem*(abs16)

Store low 16 bits to data memory, absolute addressing

sub

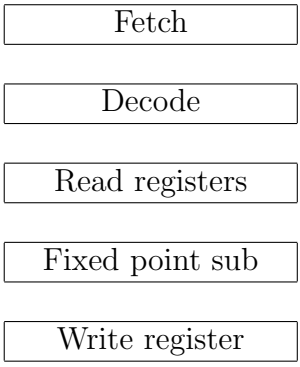
Assembler syntax:

sub rA,rB,rD

Operation:

rA − rB → rD

16-bit fixed point subtraction.



subi

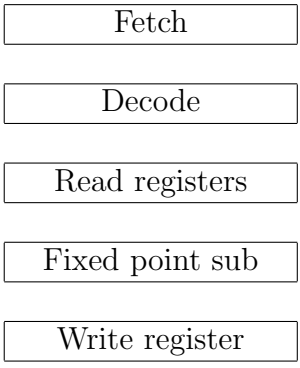
Assembler syntax:

subi #imm10,rB,rD

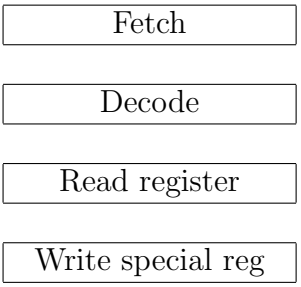
Operation:

signextend(imm10) − rB → rD

16-bit fixed point subtraction, immediate data.



wrsr



Assembler syntax:

wrsr rA,srD

Operation:

rA → srD

Set special register srD from register rA.

xor

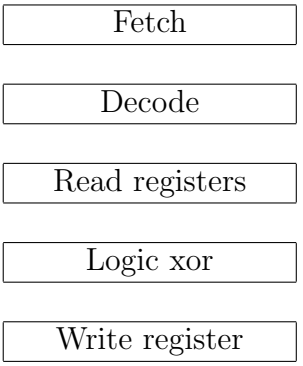
Assembler syntax:

xor rA,rB,rD

Operation:

rA $\oplus$ rB $\rightarrow$ rD

16-bit bitwise exclusive or.



xori

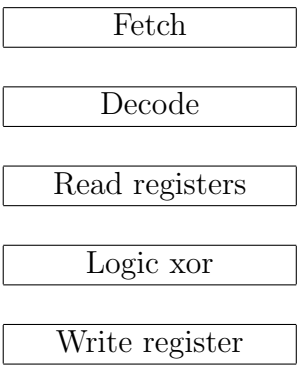
Assembler syntax:

xori #imm10,rB,rD

Operation:

signextend(imm10) $\oplus$ rB $\rightarrow$ rD

16-bit bitwise exclusive or, immediate data.



B

Instruction encoding

This chapter contains the instruction encoding.

Notation

abs16	An absolute address (16 bits).
rA	Source register A (4 bits).
rB	Source register B (4 bits).
rD	Destination register (4 bits).
srA	Special source register (4 bits).
srD	Special destination register (4 bits).
imm7	Immediate data (7 bits).
imm10	Immediate data (10 bits).
imm16	Immediate data (16 bits).
.	Don't care.

0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	**abort**	
0	0	0	1	0	0	0	0	.	.	.	.	.	.	.	.	.	.	.	.	.	.	**nop**	
0	0	0	1	0	0	0	1	.	.	.	.	.	.	.	.	.	.	.	.	.	.	**ret**	
0	0	0	1	0	0	1	0	.	.	.	.	.	.	.	.	.	.	.	rA			**jmp**	rA
0	0	0	1	0	0	1	1	.	.	.	.	.	.	.	.	.	.	.	rA			**call**	rA
0	0	0	1	0	1	0	0	abs16														**bra**	abs16
0	0	0	1	0	1	0	1	abs16														**call**	abs16
0	0	0	1	0	1	1	0	abs16														**bnbc**	abs16
0	0	0	1	0	1	1	1	abs16														**bnbs**	abs16
0	0	1	1	rD				imm16														**set**	#imm16,rD
0	1	0	0	rD				abs16														**ldi**	[abs16],rD
0	1	0	1	rB				abs16														**sti**	[abs16],rB
0	1	1	0	srD				imm16														**setsr**	#imm16,srD
0	1	1	1	rD				rB				0	0	0	0	0	.	.	.	rA		**add**	rA,rB,rD
0	1	1	1	rD				rB				0	1	0	0	0	.	.	.	rA		**sub**	rA,rB,rD
0	1	1	1	rD				rB				0	0	1	0	0	.	.	.	rA		**and**	rA,rB,rD
0	1	1	1	rD				rB				0	1	1	0	0	.	.	.	rA		**or**	rA,rB,rD
0	1	1	1	rD				rB				1	0	1	0	0	.	.	.	rA		**xor**	rA,rB,rD
0	1	1	1	rD				.	.	.	.	.	.	.	0	1	0	0	0	rA		**ltoh**	rA,rD
0	1	1	1	rD				.	.	.	.	.	.	.	0	1	0	0	1	rA		**htol**	rA,rD
0	1	1	1	rD				.	.	.	.	.	.	.	0	1	0	1	0	srA		**rdsr**	srA,rD
0	1	1	1	srD				.	.	.	.	.	.	.	0	1	0	1	1	rA		**wrsr**	rA,srD
0	1	1	1	rD				.	.	.	.	.	.	.	0	1	1	0	0	rA		**ld**	[rA],rD
0	1	1	1	rB				.	.	.	.	.	.	.	0	1	1	0	1	rA		**st**	[rA],rB
0	1	1	1	rD				.	.	.	.	.	.	.	0	1	1	1	0	rA		**nbit**	rA,rD
0	1	1	1	rD				.	.	.	.	.	.	.	0	1	1	1	1	rA		**ldf**	[rA],rD
0	1	1	1	rD				.	.	.	.	.	.	.	1	.	imm7					**seth**	#imm7,rD
1	0	0	0	rD				rB				0	0	imm10								**addi**	#imm10,rB,rD
1	0	0	0	rD				rB				0	1	imm10								**subi**	#imm10,rB,rD
1	0	0	1	rD				rB				0	0	imm10								**andi**	#imm10,rB,rD
1	0	0	1	rD				rB				0	1	imm10								**ori**	#imm10,rB,rD
1	0	0	1	rD				rB				1	0	imm10								**xori**	#imm10,rB,rD
1	0	1	0	rB				abs16														**beqz**	rB,abs16
1	0	1	1	rB				abs16														**bnez**	rB,abs16
1	1	1	0	rD				.	.	.	.	0	0	0	0	0	0	0	0	rA		**fmac**	rA,rD
1	1	1	0	rD				.	.	.	.	0	0	0	0	0	1	0	0	rA		**fmaci**	rA,rD
1	1	1	0	rD				.	.	.	.	0	0	0	0	0	1	1	0	rA		**fmulc**	rA,rD
1	1	1	1	rD				rB				.	.	.	.	.	0	0	0	rA		**fadd**	rA,rB,rD
1	1	1	1	rD				rB				.	.	.	.	.	0	0	1	rA		**fsub**	rA,rB,rD
1	1	1	1	rD				rB				.	.	.	.	.	0	1	0	rA		**fmul**	rA,rB,rD
1	1	1	1	rD				.	.	.	.	.	.	.	.	.	1	0	0	rA		**fpack**	rA,rD
1	1	1	1	rD				.	.	.	.	.	.	.	.	.	1	0	1	rA		**fint**	rA,rD

Table B.1: The instruction encoding.

C

Matlab code for the $x^{4/3}$ function

This appendix contains *matlab* code for the algorithm used in the MP3 decoder to calculate $x^{4/3}$.

```
function y = pow43(x)
% Y = POW43(X) calculates Y=X.^(4/3) using the same algorithm
% that is used in the MP3 decoder.
%
% (It is normally called for every two subband samples, but
% this matlab implementation can handle an arbitrary number
% of values in X.)

% Since this function is usually called 44100 times per second
% (44.1kHz sample rate, stereo), it has to be very fast. A few
% special cases at the beginning reduce the average execution
% time significantly. The worst case execution time can be
% estimated by calculating how many samples that must be handled
% by the general case code. This is a function of the bit rate
% (higher bit rate allows room for larger numbers).


% This is a common special case:

if x==0
    % Numbers are zero:
    y = zeros(size(x));
    return
```

```
end


% This is another common special case due to the
% way samples are encoded in the bitstream:

if abs(x)<16
    % Numbers are small, handle with lookup-table:
    y = sign(x) .* abs(x).^(4/3);
    return
end


% General case. Convert x into the internal floating point type
% (it is originally an integer). Here, the sign, exponent and
% mantissa are explicitly separated into three variables.

[m e] = log2(x);
s = sign(m);
m = 2.*abs(m);
e = e - 1;


% Use the fact that x^(4/3) = (s * 2^e * m)^(4/3) =
%  = s * (2^e)^(4/3) * m^(4/3) =(approx)
%  = s * (2^e)^(4/3) * P(m)
% where P is a fifth-order polynomial:

P = polyfit(1:0.2:2, (1:0.2:2) .^ (4/3), 5);

c1 = P(1)/P(3);
c2 = P(2)/P(4);
c3 = P(3)/P(5);
c4 = P(4)/P(6);
c5 = P(5);
c6 = P(6);
```

```
% Evaluate P(m) in way that allows parallel calculations:

m2 = m .^ 2;
m43 = ((m2 .* c1 + 1) .* m2 .* c3 + 1) .* c5 .* m + ...
        ((m2 .* c2 + 1) .* m2 .* c4 + 1) .* c6;


% This is handled by a lookup-table:
% (In the assembly language case, it also takes care
% of x==0 since this is encoded in the exponent.)

e43 = s .* (2.^e).^(4/3);


% Finally, the result:

y = e43 .* m43;
```

D

Matlab code for a fast IMDCT

This appendix contains *matlab* code for Szu-Wei Lee's IMDCT algorithm [8].

```
function y = imdct(x)
% An implementation of Szu-Wei Lee's fast IMDCT algorithm.
% 36-point IMDCT as used in MP3 decoding.

temp=dctIV(x);

% Convert 18-point DCT-IV to 36-point IMDCT
% by rearranging output and changing signs as
% appropriate
for m=0:17
    temp(36-m)=-temp(m+1);
end

for m=0:8
    y(m+1+3*9)=-temp(m+1);
end

for m=9:35
    y(m-8)=temp(m+1);
end
```

```
function X=dctIV(y)

% Scale input
for m=0:17
    temp(m+1)=y(m+1)*cos(pi*(2*m+1)/(4*18));
end

X=sdctII(temp');

% Output accumulation
for k=2:18
    X(k)=X(k)-X(k-1);
end
```

```
function X = sdctII(x)
% Divide the 18-point SDCT-II into two 9-point SDCT-II

% Even input butterfly
for m=0:8
    temp(m+1)=x(m+1)+x(18-m);
end
u=fastsdct(temp');

% Odd input butterfly and scaling
for m=0:8
    temp(m+1)=(x(m+1)-x(18-m))*2*cos(pi*(2*m+1)/(2*18));
end
v=fastsdct(temp');

% Output accumulation step
for k=1:8
    v(k+1)=v(k+1)-v(k-1+1);
end


% Return even/odd values in X
X(0+1)=u(0+1);
X(1+1)=v(0+1);

for k=1:8
    X(2*k+0+1)=u(k+1);
    X(2*k+1+1)=v(k+1);
end
```

```
function y = fastsdct(x)
x0=x(1);
x1=x(2);
x2=x(3);
x3=x(4);
x4=x(5);
x5=x(6);
x6=x(7);
x7=x(8);
x8=x(9);

d0=sqrt(3);
d1=2*cos(8*pi/9);
d2=2*cos(4*pi/9);
d3=2*cos(2*pi/9);
d4=2*sin(8*pi/9);
d5=2*sin(4*pi/9);
d6=2*sin(2*pi/9);

a1=x3+x5;
a2=x3-x5;
a3=x6+x2;
a4=x6-x2;
a5=x1+x7;
a6=x1-x7;
a7=x8+x0;
a8=x8-x0;
a9=x4+a5;

a10=a1+a3;
a11=a10+a7;
a12=a3-a7;
a13=a1-a7;
a14=a1-a3;
a15=a2-a4;
a16=a15+a8;
a17=a4+a8;
a18=a2-a8;
```

```
a19=a2+a4;

m1=-d0*a6;
m2=-d1*a12;
m3=-d2*a13;
m4=-d3*a14;
m5=-d0*a16;
m6=-d4*a17;
m7=-d5*a18;  % NOTE: The 8 seems to be missing in the paper [8]
m8=-d6*a19;

a20=x4+x4-a5;
a21=a20+m2;
a22=a20-m2;
a23=a20+m3;
a24=m1+m6;
a25=m1-m6;
a26=m1+m7;

y0=a9+a11;
y1=m8-a26;
y2=m4-a21;
y3=m5;
y4=a22-m3;
y5=a25-m7;
y6=a11-a9-a9;
y7=a24+m8;
y8=a23+m4;

y(1)=y0;
y(2)=y1;
y(3)=y2;
y(4)=y3;
y(5)=y4;
y(6)=y5;
y(7)=y6;
y(8)=y7;
y(9)=y8;
```

E

Matlab code for a fast DCT

This appendix contains *matlab* code for Byeong Lee's DCT algorithm [7].

```
function X = recursive_dct(x)
% An implementation of DCT using Lee's algorithm
% The size of x should be a power of 2
%
% Note that the output from this program is not identical
% to matlab's dct. X(1) should be scaled by 1/2 and
% all other elements of X should be scaled by 1/sqrt(2) to
% convert this dct to the same type as matlab's.

% The size of x
N=size(x);
N=N(2);

if N == 1
    X=x;
else

    % Even input butterfly
    for i=1:(N/2)
        temp(i)=x(i)+x(N-i+1);
    end

    % Recursively calculate N/2-point DCT
    resulteven=recursive_dct(temp);
```

```
    % odd input butterfly and scaling
    for i=1:(N/2)
        temp(i)=(x(i)-x(N-i+1)) * 1/(2*cos(pi/2/(N)*(2*(i-1)+1)));
    end

    resultodd=recursive_dct(temp);

    % Output butterfly
    for i=1:(N/2-1)
        temp(i)=resultodd(i)+resultodd(i+1);
    end
    temp(N/2)=resultodd(N/2);

    for i=1:(N/2)
        X(i*2-1)=resulteven(i);
        X(i*2)=temp(i);
    end
end
```

References

[1] Painter, T. and Spanias, A., "Perceptual Coding of Digital Audio", *Proceedings of the IEEE, Vol. 88, No. 4, April 2000*

[2] ISO/IEC, "Information Technology — Coding of Moving Pictures and Associated Audio for Digital Storage Media at up to About 1.5Mbit/s, Part 3: Audio", 1992

[3] "MPEG Audio Decoder Compliance", http://www.underbit.com/resources/mpeg/audio/compliance/

[4] "Layer III compliance bitstreams", ftp://ftp.tnt.uni-hannover.de/pub/MPEG/audio

[5] "ISO MP3 sources (distribution 10)", http://www.mp3-tech.org/programmer/sources/dist10.tgz

[6] Konstantinides, K., "Fast subband filtering in MPEG audio coding", *IEEE Signal Processing Letters, Vol. 1, Iss. 2, Feb 1994*

[7] Lee, B., "A new algorithm to compute the discrete cosine Transform", *IEEE Transactions on Acoustics, Speech, and Signal Processing, Vol. 32, Iss. 6, Dec 1984*

[8] Lee, S.-W., "Improved algorithm for efficient computation of the forward and backward MDCT in MPEG audio coder", *Circuits and Systems II: Analog and Digital Signal Processing, IEEE Transactions on, Vol. 48, Iss. 10, Oct 2001*

[9] Britanak, V. and Rao, K. R., "An efficient implementation of the forward and inverse MDCT in MPEG audio coding", *IEEE Signal Processing Letters, Vol. 8, No. 2, Feb 2001*

[10] "Sound Quality Assessment Material", http://www.tnt.uni-hannover.de/project/mpeg/audio/sqam/

[11] "LAME test samples",
http://lame.sourceforge.net/gpsycho/quality.html

Avdelning, Institution
Division, Department

Institutionen för Systemteknik
581 83 Linköping

Datum
Date

23 maj 2003

Språk
Language
☐ Svenska/Swedish
☒ Engelska/English
☐ _____

Rapporttyp
Report category
☐ Licentiatavhandling
☒ Examensarbete
☐ C-uppsats
☐ D-uppsats
☐ Övrig rapport
☐ _____

ISBN
—

ISRN
LITH-ISY-EX-3446-2003

Serietitel och serienummer **ISSN**
Title of series, numbering —

URL för elektronisk version
http://www.ep.liu.se/exjobb/isy/2003/3446/

Titel En hårdvarubaserad MP3-avkodare som använder flyttal med låg precision för mellanlagring

Title A hardware MP3 decoder with low precision floating point intermediate storage

Författare Andreas Ehliar, Johan Eilert
Author

Sammanfattning
Abstract

The effects of using limited precision floating point for intermediate storage in an embedded MP3 decoder are investigated in this thesis. The advantages of using limited precision is that the values need shorter word lengths and thus a smaller memory for storage.

The official reference decoder was modified so that the effects of different word lengths and algorithms could be examined. Finally, a software and hardware prototype was implemented that uses 16-bit wide memory for intermediate storage. The prototype is classified as a limited accuracy MP3 decoder. Only layer III is supported. The decoder could easily be extended to a full precision MP3 decoder if a corresponding increase in memory usage was accepted.

Nyckelord
Keywords MP3, audio, implementation, floating point

Copyright

Svenska

English